

深度学习 算法实践

(基于Theano和TensorFlow)

闫涛 周琦 编著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

作者介绍

闫涛，网名“最老程序员”。中科院苏州生物医学工程技术研究所副研究员（兼），科技部重点专项：帕金森症早期预防、“十三五”出生缺陷预防系统研究课题组成员，专注于深度学习在医学影像学诊断、医学图像分割、医学图像诊断性标注等应用方向的技术开发。CSDN 博客重度使用者，博客地址 <http://blog.csdn.net/yt7589>。北京动维康科技有限公司联合创始人、首席技术官，主持开发了移动医疗系统随诊医生。专注于移动互联网软件开发 20 年，精通主流开发技术，尤其擅长处理大容量、高并发系统的设计与实现。开源软件倡导者，本书部分代码的 GitHub 网址为 <https://github.com/yt7589/dlp/tree/master/-book>。

深度学习 算法实践

(基于Theano和TensorFlow)

闫涛 周琦 编著



电子工业出版社
Publishing House of Electronics Industry
北京·BEIJING

内 容 简 介

本书以深度学习算法入门为主要内容,通过系统介绍 Python、NumPy、SciPy 等科学计算库,深度学习主流算法,深度学习前沿研究,深度学习服务云平台构建四大主线,向读者系统地介绍了深度学习的主要内容和研究进展。本书介绍了 Python、NumPy、SciPy 的使用技巧,面向谷歌推出的开源深度学习框架 TensorFlow,向读者展示了利用 TensorFlow 和 Theano 框架实现线性回归、逻辑回归、多层感知器、卷积神经网络、递归神经网络、长短时记忆网络、去噪自动编码器、堆叠自动编码器、受限玻尔兹曼机、深度信念网络等,并将这些技术用于 MNIST 手写数字识别任务。本书不仅讲述了深度学习算法本身,而且重点讲述了如何将这些深度学习算法包装成 Web 服务。本书旨在帮助广大工程技术人员快速掌握深度学习相关理论和实践,并将这些知识应用到实际工作中。

本书可以作为各类深度学习培训班的教材,也可以作为全国高等工科院校“深度学习”课程的教材,还可以作为广大人工智能、深度学习领域工程技术人员的参考书。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有,侵权必究。

图书在版编目(CIP)数据

深度学习算法实践:基于 Theano 与 TensorFlow / 闫涛,周琦编著. —北京:电子工业出版社,2018.4

ISBN 978-7-121-33793-2

I. ①深… II. ①闫… ②周… III. ①人工智能—算法—研究 IV. ①TP18

中国版本图书馆 CIP 数据核字(2018)第 041449 号

策划编辑:付 睿

责任编辑:牛 勇

特约编辑:赵树刚

印 刷:

装 订:

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱

邮编:100036

开 本:787×1092 1/16 印张:36.5 字数:934.4 千字

版 次:2018 年 4 月第 1 版

印 次:2018 年 4 月第 1 次印刷

定 价:109.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888, 88258888。

质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式:010-51260888-819, faq@phei.com.cn。

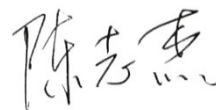
推荐序

《深度学习算法实践（基于 Theano 和 TensorFlow）》针对深度学习初学者的需求，详细讲解了深度学习典型算法的数学原理，给出了基于 TensorFlow 和 Theano 的算法实现，并以手写数字识别、图像标注、文本生成等为例，演示了深度学习算法的典型应用。作者立足于引导读者从解决问题的思路出发，层层剖析，逐步开发出实用的深度学习系统。通过阅读本书，在熟练掌握深度学习基本数学原理的基础上，读者不仅可以直接将书中内容用于项目实践，而且可以跟踪理解深度学习的最新进展。

自 2017 年下半年以来，深度学习又有了一些新进展，如注意力机制、生成式对抗网络、胶囊网络等，虽然本书还没有将其详细纳入，但是相信读者基于本书的知识架构，通过阅读相关论文及文献，理解并掌握这些算法并不困难。

当前，人工智能、深度学习技术的发展可谓一日千里，需要时刻跟踪业界的最新进展，才能保证自己的知识结构跟上业界发展步伐。作者拥有较深的理论造诣和丰富的实践经验，希望本书能够帮助读者掌握完整的知识体系，拥有较强的动手能力，成为人工智能、深度学习领域的学习型和实践型人才。

中国工程院院士



2018 年 3 月 9 日

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误：**您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方 [读者评论](#) 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/33793>



前言

2016 年上半年，随着 AlphaGo 战胜围棋世界冠军李世石，深度学习技术迅速进入大众的视野，成为热门技术。无论是科技领域还是创业投资领域，深度学习技术都受到了前所未有的关注，面向深度学习技术的初创公司不断刷新融资数额的纪录。

随着深度学习技术的流行，市场上对于深度学习人才的需求量激增。但是，由于深度学习技术涉及高等数学、线性代数、数理统计和机器学习相关知识，故学习曲线比较陡峭。目前国内外著名大学深度学习相关专业硕博士、国家重点实验室学生，都被 BAT 等一线互联网公司抢光了，人才大战有愈演愈烈之势。普通公司和初创公司，在这场人才大战中处于劣势，面临着越来越严重的人才荒。

随着深度学习技术的深入应用，企业对深度学习技术人才的渴求是正常的，但是当前市场上对深度学习技术人才的需求是非理性的。一方面，大家疯抢的深度学习人才都是著名院校的硕博士和重点实验室的研究员，但是这部分人所受的训练和精通的领域是做算法研究，而企业的目的是实际应用，二者在很大程度上是不匹配的，最后可能是企业下了血本却没有产生效益；另一方面，对于很多深度学习应用来说，对项目本身业务的理解，比对深度学习算法的理解要重要得多，而由于业务专家不懂深度学习技术，致使很多好的应用领域没有合适的人才来做。

编写本书的目的就是解决上述问题，希望读者可以通过对本书的学习，迅速掌握深度学习的理论框架和知识体系，具备在自己的专业领域内应用深度学习技术的能力，同时还具备跟踪深度学习领域最新进展的能力，能够独立复现顶级期刊文章中介绍的新方法和新理论。

本书内容及知识体系

第一部分为深度学习算法概述，包括第 1 章。

第 1 章简单介绍神经网络和深度学习发展史、现状和发展趋势，介绍并比较了深度学习开源框架，还介绍了开源框架的选择标准。

第二部分为深度学习算法基础，主要讲述深度学习算法中比较成熟的算法，包括第 2 章到第 7 章。

第 2 章介绍 Python 开发环境的搭建、NumPy 的使用、Theano 的安装和使用，并用本章介绍的知识实现一个简单的线性回归算法的程序。

第 3 章讲述逻辑回归算法的数学推导过程，并且讲述了通用学习模型的相关知识，还利用逻辑回归算法对 MNIST 手写数字识别数据集进行训练和识别。

第 4 章讲述多层感知器模型算法推导过程，以及数值计算与向量表示方式，并利用多层感知器模型对 MNIST 手写数字识别数据集进行训练和识别。

第 5 章讲述卷积神经网络的数学原理，详细讲解卷积神经网络的层间稀疏连接、权值共享和最大池化等技术，并利用卷积神经网络模型对 MNIST 手写数字识别数据集进行训练和识别。

第 6 章讲述递归神经网络的数学原理，以字符 RNN 网络为例，向读者演示了简单的计算机写作系统。而且，以微软图像标注数据集为例，以测试驱动开发的形式，向读者介绍利用递归神经网络做图像标注的典型应用。

第 7 章讲述长短时记忆网络的网络架构和数学原理，并以大型影评数据集为例，采用长短时记忆网络进行情感计算。

第三部分为深度学习算法进阶，主要讲述深度学习算法中比较前沿的算法，包括第 8 章到第 11 章。

第 8 章讲述自动编码器的数学原理，重点介绍实际中应用较多的去噪自动编码器和稀疏自动编码器，并以去噪自动编码器为例，对 MNIST 手写数字识别数据集进行特征提取。

第 9 章讲述将去噪自动编码器进行分层训练，组合成堆叠去噪自动编码器，并将堆叠去噪自动编码器用于 MNIST 手写数字识别任务。

第 10 章讲述受限玻尔兹曼机的数学原理，并将其用于 MNIST 手写数字识别任务。

第 11 章讲述深度信念网络的数学原理，以及其与受限玻尔兹曼机的关系，并将其用于 MNIST 手写数字识别任务。

第四部分为机器学习基础，主要讲述一些基础的机器学习算法，包括第 12 章和第 13 章。

第 12 章讲述生成式学习的基础理论，并将高斯判别分析用于癌症判别，将朴素贝叶斯算法用于垃圾邮件过滤。

第 13 章简单介绍支撑向量机算法的数学原理。

第五部分为深度学习平台 API，这部分讲述将深度学习算法包装成深度学习服务云平台的技术，包括第 14 章和第 15 章。

第 14 章介绍 Python 的 Web 开发环境及开发技术。

第 15 章应用 Web 开发技术，将前面介绍的多层感知器模型包装成 RESTful 服务，用户可以通过网页上传图片文件，并得到识别后的结果。

由于篇幅所限，书中很多例子只给出了部分代码，这些代码对于理解算法的实现原理是足够的，但是考虑到代码的完整性，我们将书中绝大部分例程都上传到了 GitHub 的开源项目 <https://github.com/yt7589/dlp.git>，书中的代码放在 book/chp**目录下，这些代码在 Ubuntu 16.04+Python3.6+TensorFlow1.2 和 Windows+Anaconda+TensorFlow1.2 下均可正常运行。读者可以下载相关源码，通过运行这些源码加深对书中内容的理解。

目 录

第一部分 深度学习算法概述

第 1 章 深度学习算法简介	2
1.1 神经网络发展简史	2
1.1.1 神经网络第一次兴起	3
1.1.2 神经网络沉寂期（20 世纪 80 年代—21 世纪）	4
1.1.3 神经网络技术积累期（20 世纪 90 年代—2006 年）	5
1.1.4 深度学习算法崛起（2006 年至今）	8
1.2 深度学习现状	10
1.2.1 传统神经网络困境	10
1.2.2 深度多层感知器	12
1.2.3 深度卷积神经网络	14
1.2.4 深度递归神经网络	15
1.3 深度学习研究前瞻	16
1.3.1 自动编码器	17
1.3.2 深度信念网络	18
1.3.3 生成式网络最新进展	19
1.4 深度学习框架比较	20
1.4.1 TensorFlow	20
1.4.2 Theano	21
1.4.3 Torch	22
1.4.4 DeepLearning4J	23
1.4.5 Caffe	23
1.4.6 MXNet	24
1.4.7 CNTK	27
1.4.8 深度学习框架选型指导原则	27
1.5 深度学习入门路径	28
1.5.1 运行 MNIST	28

1.5.2	深度学习框架的选择.....	29
1.5.3	小型试验网络.....	33
1.5.4	训练生产网络.....	33
1.5.5	搭建生产环境.....	34
1.5.6	持续改进.....	35

第二部分 深度学习算法基础

第 2 章	搭建深度学习开发环境.....	38
2.1	安装 Python 开发环境	38
2.1.1	安装最新版本 Python	38
2.1.2	Python 虚拟环境配置	39
2.1.3	安装科学计算库.....	40
2.1.4	安装最新版本 Theano.....	40
2.1.5	图形绘制.....	40
2.2	NumPy 简易教程	43
2.2.1	Python 基础	43
2.2.2	多维数组的使用.....	51
2.2.3	向量运算.....	58
2.2.4	矩阵运算.....	60
2.2.5	线性代数.....	62
2.3	TensorFlow 简易教程.....	68
2.3.1	张量定义.....	69
2.3.2	变量和 placeholder.....	69
2.3.3	神经元激活函数.....	71
2.3.4	线性代数运算.....	72
2.3.5	操作数据集.....	74
2.4	Theano 简易教程.....	77
2.4.1	安装 Theano.....	77
2.4.2	Theano 入门.....	78
2.4.3	Theano 矩阵相加.....	79
2.4.4	变量和共享变量.....	80
2.4.5	随机数的使用	84
2.4.6	Theano 求导.....	84
2.5	线性回归.....	86
2.5.1	问题描述.....	86
2.5.2	线性模型.....	88

2.5.3	线性回归学习算法.....	89
2.5.4	解析法.....	90
2.5.5	Theano 实现.....	93
第 3 章	逻辑回归	100
3.1	逻辑回归数学基础.....	100
3.1.1	逻辑回归算法的直观解释.....	100
3.1.2	逻辑回归算法数学推导.....	101
3.1.3	牛顿法解逻辑回归问题.....	103
3.1.4	通用学习模型.....	106
3.2	逻辑回归算法简单应用.....	113
3.3	MNIST 手写数字识别库简介	124
3.4	逻辑回归 MNIST 手写数字识别	126
第 4 章	感知器模型和 MLP	139
4.1	感知器模型.....	139
4.1.1	神经元模型.....	139
4.1.2	神经网络架构.....	143
4.2	数值计算形式.....	144
4.2.1	前向传播.....	144
4.2.2	误差反向传播.....	145
4.2.3	算法推导.....	147
4.3	向量化表示形式.....	152
4.4	应用要点.....	153
4.4.1	输入信号模型.....	154
4.4.2	权值初始化.....	155
4.4.3	早期停止.....	155
4.4.4	输入信号调整.....	156
4.5	TensorFlow 实现 MLP	156
第 5 章	卷积神经网络	174
5.1	卷积神经网络原理.....	174
5.1.1	卷积神经网络的直观理解.....	174
5.1.2	卷积神经网络构成.....	177
5.1.3	卷积神经网络设计.....	191
5.1.4	迁移学习和网络微调.....	193
5.2	卷积神经网络的 TensorFlow 实现.....	195
5.2.1	模型搭建.....	197

5.2.2	训练方法.....	203
5.2.3	运行方法.....	208
第 6 章	递归神经网络	212
6.1	递归神经网络原理.....	212
6.1.1	递归神经网络表示方法	213
6.1.2	数学原理.....	214
6.1.3	简单递归神经网络应用示例	219
6.2	图像标记.....	226
6.2.1	建立开发环境.....	226
6.2.2	图像标记数据集处理	227
6.2.3	单步前向传播.....	229
6.2.4	单步反向传播.....	231
6.2.5	完整前向传播.....	234
6.2.6	完整反向传播.....	236
6.2.7	单词嵌入前向传播.....	239
6.2.8	单词嵌入反向传播.....	241
6.2.9	输出层前向/反向传播.....	243
6.2.10	输出层代价函数计算.....	245
6.2.11	图像标注网络整体架构	248
6.2.12	代价函数计算.....	249
6.2.13	生成图像标记.....	255
6.2.14	网络训练过程.....	258
6.2.15	网络持久化.....	265
第 7 章	长短时记忆网络.....	269
7.1	长短时记忆网络原理.....	269
7.1.1	网络架构.....	269
7.1.2	数学公式.....	272
7.2	MNIST 手写数字识别	274

第三部分 深度学习算法进阶

第 8 章	自动编码器.....	286
8.1	自动编码器概述.....	286
8.1.1	自动编码器原理.....	287
8.1.2	去噪自动编码器.....	287

8.1.3 稀疏自动编码器.....	288
8.2 去噪自动编码器 TensorFlow 实现.....	291
8.3 去噪自动编码器的 Theano 实现.....	298
第 9 章 堆叠自动编码器.....	307
9.1 堆叠去噪自动编码器.....	308
9.2 TensorFlow 实现.....	322
9.3 Theano 实现.....	341
第 10 章 受限玻尔兹曼机.....	344
10.1 受限玻尔兹曼机原理.....	344
10.1.1 网络架构.....	344
10.1.2 能量模型.....	346
10.1.3 CD-K 算法.....	351
10.2 受限玻尔兹曼机 TensorFlow 实现.....	353
10.3 受限玻尔兹曼机 Theano 实现.....	362
第 11 章 深度信念网络	381
11.1 深度信念网络原理.....	381
11.2 深度信念网络 TensorFlow 实现.....	382
11.3 深度信念网络 Theano 实现.....	403

第四部分 机器学习基础

第 12 章 生成式学习	420
12.1 高斯判别分析.....	422
12.1.1 多变量高斯分布.....	422
12.1.2 高斯判决分析公式.....	423
12.2 朴素贝叶斯.....	436
12.2.1 朴素贝叶斯分类器.....	436
12.2.2 拉普拉斯平滑.....	439
12.2.3 多项式事件模型.....	441
第 13 章 支撑向量机.....	444
13.1 支撑向量机概述.....	444
13.1.1 函数间隔和几何间隔.....	445
13.1.2 最优距离分类器.....	448
13.2 拉格朗日对偶.....	448

13.3	最优分类器算法.....	450
13.4	核方法.....	453
13.5	非线性可分问题.....	455
13.6	SMO 算法.....	457
13.6.1	坐标上升算法.....	458
13.6.2	SMO 算法详解.....	458

第五部分 深度学习平台 API

第 14 章	Python Web 编程	462
14.1	Python Web 开发环境搭建	462
14.1.1	CherryPy 框架	463
14.1.2	CherryPy 安装	463
14.1.3	测试 CherryPy 安装是否成功	464
14.2	最简 Web 服务器	465
14.2.1	程序启动.....	465
14.2.2	显示 HTML 文件	466
14.2.3	静态内容处理.....	468
14.3	用户认证系统.....	471
14.4	AJAX 请求详解	473
14.4.1	添加数据.....	474
14.4.2	修改数据.....	476
14.4.3	删除数据.....	478
14.4.4	REST 服务实现.....	479
14.5	数据持久化技术.....	487
14.5.1	环境搭建.....	487
14.5.2	数据库添加操作.....	488
14.5.3	数据库修改操作.....	489
14.5.4	数据库删除操作.....	490
14.5.5	数据库查询操作.....	491
14.5.6	数据库事务操作.....	492
14.5.7	数据库连接池.....	494
14.6	任务队列.....	499
14.7	媒体文件上传.....	502
14.8	Redis 操作	504
14.8.1	Redis 安装配置.....	504
14.8.2	Redis 使用例程.....	505

第 15 章 深度学习云平台.....	506
15.1 神经网络持久化.....	506
15.1.1 数据库表设计.....	506
15.1.2 整体目录结构.....	511
15.1.3 训练过程及模型文件保存.....	512
15.2 神经网络运行模式.....	528
15.3 AJAX 请求调用神经网络.....	531
15.3.1 显示静态网页.....	531
15.3.2 上传图片文件.....	540
15.3.3 AJAX 接口.....	543
15.4 请求合法性验证.....	545
15.4.1 用户注册和登录.....	546
15.4.2 客户端生成请求.....	553
15.4.3 服务器端验证请求.....	555
15.5 异步结果处理.....	557
15.5.1 网页异步提交.....	557
15.5.2 应用队列管理模块.....	559
15.5.3 任务队列.....	560
15.5.4 结果队列.....	561
15.5.5 异步请求处理流程.....	562
15.6 神经网络持续改进.....	563
15.6.1 应用遗传算法.....	563
15.6.2 重新训练.....	564
15.6.3 生成式对抗网络.....	565
后 记.....	567
参考文献.....	568

第一部分 深度学习算法概述

□ 深度学习算法简介

第 1 章

深度学习算法简介

1.1 神经网络发展简史

人工神经网络（ANN）是由许多简单的、相互连接的处理单元组成的，这些处理单元被称为神经元。人工神经网络是对生物神经系统进行仿生设计的结果，因此人工神经元与生物神经元有许多相似之处，每个神经元都具有若干个输入信号，神经元经过处理（通常是非线性处理）产生一个实数值的输出，通过输出突触传递给周围的神经元。整个人工神经网络，通过接收外界的输入信号，经过神经元的协同运算，产生人们希望的结果。从过去几十年的发展历史来看，相对于其他方法而言，人工神经网络对于很难用显性知识表示的领域，例如计算机视觉、模式识别、人脸识别、语音识别、视频识别等领域，具有非常大的优势。

神经网络的学习过程主要是指通过发现合适的神经元间的连接权值，来使整个神经网络表现出我们希望的行为，如自动驾驶、图像识别等。根据需要处理的问题，以及神经元间连接方式的不同，神经网络的决策过程可能包括很长的因果链，或者说多个计算阶段。在每个阶段中，神经网络都会自动汇总网络的激活信号，经过特定的非线性变换，为下一阶段的运算做好准备，并最终产生我们需要的结果。深度学习网络正是这样一种结构，它完美地体现了分阶段任务的特性，因此在实际应用中取得了令人刮目相看的成就。

我们知道，浅层神经网络在很久以前就存在了，20 世纪 60、70 年代，就有多层非线性神经网络应用的例子。而且，基于监督学习的梯度下降方法，对于连续可微的多重函数，人们发现了反向传播算法（BP），并且在 1981 年成功将其应用于神经网络，使得基于这种算法的前馈神经网络在 20 世纪 80 年代迅速流行起来。然而在 20 世纪 80 年代末期，人们发现将 BP 算法应用于深度网络会出现非常大的问题，鉴于即使只有一个隐藏层神经网络，

只要隐藏层神经元数目足够多，也可以拟合任意函数，因此当时人们普遍认为浅而宽的神经网络具有更好的性能。但是对于深度网络的研究并没有停止，尤其是研究人员将非监督学习引入深度学习网络之后，随着算法和神经元激活函数方面的一系列改进，使得深度学习网络在一系列模式识别竞赛中获奖，尤其是 2011 年，在某些特定领域，深度学习网络的模式识别能力甚至超过了人类。与此同时，深度学习在商业应用方面也取得了巨大的成功，例如在图像识别、语音识别和自然语言处理方面，取得了一系列令人瞩目的成绩，这一趋势一直延续至今。

在过去十几年中，前向神经网络（FNN）和递归神经网络（RNN）都赢得了一系列的算法竞赛。从某种意义上来说，递归神经网络是最深的神经网络，拥有比前向神经网络更强的计算能力，因为递归神经网络将序列信号分析和信息并行处理，以一种自然、高效的方式结合在一起。配合当前不断增加的计算能力，可以预见递归神经网络将拥有越来越重要的地位。

然而，神经网络的发展并不是一帆风顺的，中间经历数次大起大落，本节我们将简要回顾一下神经网络的发展史，通过介绍深度学习之前神经网络所遇到的困境，使读者对深度学习的背景有一个清晰的了解，这对于理解深度学习发展趋势是非常重要的，因为历史总是惊人的相似。同时，我们也应该看到，深度学习并不是人工神经网络发展的终点，其只是人工神经网络发展的一个特定阶段，一定会有更先进的技术超越当前的深度学习技术，因为深度学习网络目前只是将学习过程进行了分解，同时用非监督学习来辅助监督学习，并没有从根本上解决人工神经网络中欠拟合（Under Fitting）和过拟合（Over Fitting）等泛化问题。如果可以在强化学习和非监督学习方面取得突破，理论上可以产生更好的学习算法。

1.1.1 神经网络第一次兴起

现代神经网络起源于 20 世纪 40 年代，早期典型的神经网络模型有 Hebb 在 1949 年提出的非监督学习模型，以及 1958 年 Rosenblatt 提出的基于监督学习的感知器模型，其后出现了一系列监督学习和非监督学习模型。总体来讲，由于这一阶段计算能力有限，神经网络模型都比较简单，而且缺乏成功的实际应用。

最早的深度学习网络模型出现在 20 世纪 60~70 年代，Ivakhnenko 等人提出了 GMDH 网络模型，在这种网络模型下，神经元的激活函数为 Kolmogorov-Gabor 多项式。在给定一个训练集的情况下，先通过回归分析增加神经网络的层数，再通过单独的验证集对网络拓扑结构进行剪裁。综上所述，GMDH 神经网络的网络层数和每层的神经元数，都可以通过学习来进行调整，并且针对特定的问题，网络拓扑结构也将不同。可以说，GMDH 神经网络是第一种开放的、分层学习的神经网络模型，当时最深的网络可以达到 8 层。

在 20 世纪 60~70 年代，神经生理学家们发现，在猫的视觉皮层中有两类细胞，一类是简单细胞，另一类是复杂细胞。简单细胞对图像中的细节信息更敏感，例如边缘等；

而复杂细胞具有空间不变性，可以处理目标远近、旋转、放缩及遮挡等情况。这一发现直接导致了卷积神经网络的提出。在卷积神经网络中，层间神经元不是全连接的，而是连接于其接收域。接收域形成了 2D 区域，卷积滤波器会依次对这个 2D 区域进行滤波操作，滤波结果经过向下采样，形成新的输出信号，传递给下一层。由于大规模采用权值共享机制，可以说卷积神经网络（CNN）是第一种可以应用于实践中的深度学习网络。在后面的讨论中我们可以看到，当时的卷积神经网络，与当代获得模式识别竞赛大奖的卷积神经网络（MPCNN）还是非常相似的。

在复杂、非线性、可微分的多层神经网络中，通过梯度下降算法来最小化误差函数，在 20 世纪 60 年代就得到了广泛的应用。在神经网络的权值空间中，通过反复使用链式微分法则，可以采用最速梯度下降算法来进行神经网络学习。但是这些算法只使用了 Jacobian 矩阵计算，没有考虑到层间的连接特性和网络的稀疏性，因此在 1969 年 Minsky 和 Papert 指出了这些神经网络的局限性，在此后很长时间内，神经网络的研究和应用处于低潮期。

尽管没有明确提及神经网络，但 Linnainmaa 在 1970 年的硕士论文中，提出了在任意、离散、潜在稀疏的计算网络中，可以采用显式高效的误差反向传播算法。将误差反向传播算法（BP 算法）用于神经网络，始于 Werbos 在 1981 年的文章。在 1986 年，实验证明了神经网络隐藏层的内部表现方式，它可以使 BP 算法在多层神经网络得到高效应用，这一研究成果使 BP 算法迅速在神经网络研究中占据了主导地位。

1.1.2 神经网络沉寂期（20 世纪 80 年代—21 世纪）

在 20 世纪 80 年代中后期，人们发现 BP 算法在神经网络应用中存在很多问题，并不是能解决所有问题的“灵丹妙药”。当时所有实用的神经网络，都是只有很少隐藏层的神经网络，因为人们发现，多层神经网络在实际应用中的效果并不理想。与此同时，Kolmogorov、Hecht、Hornik 证明，只有一个隐藏层的神经网络系统，只要隐藏层神经元足够多，就能以任意精度拟合任意连续函数。当时人们普遍认为，尽管 BP 算法可以用于多层神经网络，但是 BP 算法只有在浅层神经网络应用中才有理想的效果。

研究者们想出了各种方法来解决 BP 算法在多层神经网络中的问题，总体来讲，这些方法虽然有所改进，但是都没有取得理想的效果。其中出现了两种非常有意思的网络，一种是神经热能交换器网络，另一种是 BP 逻辑网络。神经热能交换器网络由 Schmidhuber 在 1990 年提出，由两个相反方向的前馈神经网络组成。输入先从第一个神经网络逐层向上传播，到达输出层，可以类比于加热过程；然后将这些输出信号作为第二个神经网络的输入信号，逐层向下传播，最后到达输入层，重新变为输入信号，可以类比于冷却过程。

在 1989 年，Shavlik 和 Towell 提出了将领域知识中的命题逻辑利用解释器生成深层前馈神经网络，这种神经网络的深度可以用最长命题逻辑链条来定义。这种神经网络的权值可以通过领域知识中的命题逻辑来初始化，并采用 BP 算法对神经网络连接权值进行微调，用来解决领域知识中的不确定性问题。这种神经网络的一个优点是可解释性，因为传统的

神经网络对于人们来讲是一个黑盒，人们只能得到结果，不能对结果进行解释。但是这种神经网络不仅可以得到正确的结果，而且可以给出合理的解释。

在前馈神经网络中，当时最流行的算法是最速梯度下降算法。为了提高最速梯度下降算法的效率，人们想出了各种改进方法，包括牛顿法、准牛顿法、共扼梯度法等。

在所有算法中，BP 算法不仅收敛速度较慢，而且更为严重的是容易陷入局部最小值点。为了解决这一问题，研究者们提出了动量项的概念，即针对 BP 算法容易进入局部最小值点，或者非常平坦的区域的特点，在 BP 算法中引入动量项，试图使其跳出局部最小值点或局部平坦区域，增加找到全局最小值点的可能性，取得了较好的效果。

有一些研究者，只考虑误差求导中的符号，而不关心其具体数值，于是提出了 Rprop 算法和更稳定的 iRprop+ 算法。

在 BP 算法中，学习率是一个重要的参数，如果取值太小，则收敛速度慢；如果取值太大，则会产生波动，不容易找到最优解。通常学习率为一个特别小的正数，如小于 0.1 的正数。但是一些研究者发现，在 BP 算法前期，由于需要调整的幅度较大，适合比较大的学习率。而在 BP 算法后期，则需要较小的学习率来进行微调，找到神经网络的最优解，所以提出了自适应学习率的算法。

Hochreiter 1991 年发表的论文，可以说是深度学习的一个里程碑式的事件。在这篇论文中，Hochreiter 指出，随着神经网络的深度加深，BP 算法会出现梯度消失或梯度爆炸问题。这就是说，随着层数增多，BP 算法的误差函数的导数会逐渐衰减，直到消失；或者会逐渐变大，直到超出取值范围。此后，在研究者对深度学习算法的研究中，都是采用各种方法来解决这一问题的。

以上我们主要讨论的是神经网络中的监督学习，其实非监督学习的神经网络在此期间也得到了长足发展，同时研究者们认识到，可以利用非监督学习的神经网络来协助监督学习神经网络更好地完成任务。多数非监督学习方法都是根据信息理论优化信息熵，自动找到输入信号的分布式、稀疏式表示形式，因此这类网络又被称为特征提取器。例如，一些非监督学习网络，可以作为边缘识别器或 Gabor 滤波器，这些特征提取器与生物系统中的视觉预处理阶段非常类似。

1.1.3 神经网络技术积累期（20 世纪 90 年代—2006 年）

非监督学习神经网络可以将输入信号转换为一种更适合算法处理的形式。在深度学习中，非监督学习的一个重要任务就是消除冗余，在理想情况下，给定一组输入信号，非监督学习算法可以找到这组输入信号的因子分解码，因子之间相互独立。在实际应用中，这种因子分解码表示方式可能是稀疏的，同时有利于进行后续处理，如数据压缩、加速后续应用 BP 算法的效率、简化贝叶斯分类器等。

在早期多数非监督前馈网络中只有一个隐藏层，但是也有一些深度非监督前馈网络存在。深层非监督学习网络包括：层级自组织 Kohonen 网络、自组织树算法（SOTA）、堆叠

自动编码器（AE）。其中应用比较广泛的是堆叠自动编码器，经典的单个自动编码器有 3 层，输入的节点数与输入信号的维度相同，隐藏层节点数少于输入信号节点数，输出层神经元节点数与输入层相同。信号由输入层输入，经过隐藏层，通常隐藏层比输入层有更少的节点，对输入信号进行压缩，并传输到输出层。自动编码器算法就是保证网络输出层信号与输入层信号尽可能相等，这样隐藏层的输出就可以视为输入信号的压缩编码。在多层自动编码器中，会将前一层自动编码器的隐藏层作为输入，接入下一层的自动编码器的输入层，如此循环下去，组成一个层叠的多层自动编码器网络。在这种网络中，由于可以逐层对自动编码器进行训练，故可以有效地解决深度神经网络训练困难的问题，因此在实际中得到了广泛应用。

除堆叠自动编码器之外，还有一些神经网络也采用类似的方法。例如，分层 Kohonen 网络和受限玻尔兹曼机（RBM），受限玻尔兹曼机得到了广泛的应用，目前比较流行的深度信念网络（DBN）就是通过层叠受限玻尔兹曼机来组成的。这种先通过竞争性非监督学习算法对深度网络进行预训练，再通过 BP 算法对网络权值空间进行微调的方法，在实际应用中取得了巨大的成功。

最早的实用的深度学习网络，是 Schmidhuber 在 1992 年提出的，该网络由数百层组成，是由经过非监督学习预训练的递归神经网络组成的。对单个 RNN 网络进行非监督学习训练，预测下一个输入信号，当训练一段时间之后，仅将出错的输入信号输入到上层递归神经网络中进行训练，以此类推，直到训练到最上层。这样就形成了一个对输入信号逐层压缩的网络，越往上层走，压缩比例越高，冗余越低。这种网络目前依然是一种重要的神经网络，被称为神经层级时序记忆（HTM），与堆叠自动编码器和深度信念网络类似。

在卷积神经网络中引入最大池化层（MP），可以极大地提高卷积神经网络的性能。在这种网络中，首先对输入信号进行卷积操作，对卷积结果进行向下采样，选取卷积结果中的最大值，从而形成新一层的输入。在实际应用中，采用权值共享、稀疏表示、卷积操作、最大池化的卷积神经网络（MPCNN），配合 BP 学习算法微调，成为当前图像识别中的主流算法，也是近些年来在 MNIST 手写字母识别和 ImageNet 中获奖算法的基础。

在前馈神经网络发展的同时，递归神经网络也得到了长足发展，其中具有里程碑意义的网络就是基于监督学习的长短时记忆网络（LSTM）。对于普通的递归神经网络而言，虽然输入信号在隐藏层产生的输出会再次传回隐藏层，但是经过数个输入信号之后，原来输入信号对网络的影响基本消失了，可以说这种网络不具有长期记忆的特性。而有很多应用，则需要递归神经网络具有长期记忆的特性，因此研究者们提出长短时记忆网络。在这种网络中，隐藏层的神经元被记忆块所取代，记忆块的核心是记忆细胞，并且具有输入门、输出门、遗忘门。当输入门打开，输出门和遗忘门关闭时，网络开始学习；当输入门关闭时，网络就可以保存该输入的记忆，直到遗忘门打开，网络忘记之前的记忆，这样可以开始重新学习。

长短时记忆网络有很多种变形，可以用于解决不同的问题。长短时记忆网络可以用来解决很多其他深度学习网络无法解决的问题，如被噪声分割的时序信号、长时间高精度保存实数数值、对连续输入流进行算术运算、在强噪声背景下识别短暂的有用信号等。

长短时记忆网络还被应用于上下文无关语言 (CFL) 中, 而原有的隐马尔可夫模型和 FSA 却不能表示这类问题。长短时记忆网络不仅成功应用于与上下文无关的小语料训练集, 提取出通用规则, 而且成功应用于大语料训练集。而其他方法通常可以在小语料训练集上取得成功, 但是不能在大语料训练集上取得成功。

在上下文敏感语言问题中, 长短时记忆网络也取得了很好的效果。长短时记忆网络最少只需要 30 个最短样本 ($n \leq 10$), 就可以成功预测前缀长度大于 1000 的连续序列。这表示, 当网络训练完成后, 可以读取大于 30000000 个符号 (逐个字符读入), 最终可以识别出很微小的差异, 例如可以判断 $a(10,000,000) b(10,000,000) c(10,000,000)$ 是正确的, 而 $a(10,000,000) b(9,000,000) c(10,000,000)$ 是错误的。

双向递归神经网络 (BRNN) 用于处理当前任务同时需要上下文的情况, 例如当手写字母识别时, 如果知道前面的字符也知道后面的字符, 比仅知道前面的字符的准确率会高很多。所以双向递归神经网络可以用于预先知道整个序列的场合, 对于语音识别等领域, 由于存在句子间停顿, 可以在接收到完整的一句话后再开始进行语音识别, 因此也可以运用双向递归神经网络。双向递归神经网络实际上是由两个共享输出层的递归神经网络构成的, 第一个递归神经网络处理正向输入信号, 第二个神经网络处理反向输入信号, 二者在输出层进行叠加处理。双向递归神经网络应用最成功的场景是蛋白质二级结构预测、蛋白质交互模型和有机分子性质等方面。

对于无法确定输入信号边界的应用, 在输入信号很长或维度很高的情况下, 研究者们提出了递归神经网络专用的连接序列分类 (CTC) 输出层, 利用梯度下降相关方法, 确定最优连接权值, 最大化误差信号, 这样可以成功应用于模式识别和图像分割等领域。

当前递归神经网络被认为最强大的神经网络之一, 各种新型网络层出不穷。例如, 从本质上来说, 输入信号是一维的时序信号, 而图像应用是二维的时序信号, 视频应用是三维的时序信号, 核磁共振等是四维的时序信号, 所以研究者提出了多维递归神经网络 (MDRNN), 这种网络是由多个相对独立的双向长短时记忆网络组成的。由于图像和视频数据量极为庞大, 相应的网络计算量也非常巨大, 因此有研究者提出分层递归神经网络的概念, 首先对输入信号进行逐层压缩, 然后分别采用递归神经网络来进行处理。

为了衡量各种神经网络算法的效果, 与其他领域类似, 神经网络领域也有很多测试样本集, 例如 MNIST 的手写数字识别、ImageNet 大型图像集识别、cifar10/100 小尺寸图片识别等。这些测试集每年都会进行竞赛, 来比较各个研究小组算法的效果, 每年各种竞赛的优胜者基本代表了该领域的最高水平, 因此也会受到业界的特别关注。

在 2000 年之前, 这些竞赛的获奖者多数是非神经网络的研究者, 比如有很多研究者利用支持向量机 (SVM) 算法赢得了一系列的比赛。支持向量机是一种改良的线性分类器判决算法, 由于其相对简单, 并且具有足够高的分类精度, 因此在线性可分和一些非线性可分的分类问题中得到了广泛应用。

神经网络在此期间也有很多有益的探索, 例如 2006 年由 Neal 提出的基于贝叶斯理论的神经网络赢得了 NIPS 2003 特征抽取大赛的冠军。这个神经网络具有两个隐藏层, 还不是深度学习网络。

1.1.4 深度学习算法崛起（2006 年至今）

在所有神经网络中，卷积神经网络在一系列图像识别竞赛中取得了压倒性的胜利。例如，Simard 在 2003 年利用 LeCun 的卷积神经网络（该网络采用卷积神经网络结构）和误差反向传播算法进行训练，在 MNIST 手写数字识别中达到了 0.4% 的误差率，这个识别精度甚至高于人类的识别准确度。但是 Simard 采用了利用训练样本进行变形处理的方式，增加了训练样本的数量。同样，Simard 在 2003 年，利用标准的前向 BP 网络也达到了 0.7% 的误差率。但是这些网络的层数还比较少，不能称之为深度学习网络。

在图像理解方面，深层神经网络取得了一定的进展。在 2004 年，Behnke 在前馈网络的隐藏层中加入了反馈连接，采用改进的反向传播 Rprop 算法，在图像识别方面取得了较好的效果。

在此之后，神经网络在图像模式识别方面的应用突破，主要集中在利用非监督学习辅助监督学习领域。尽管深度学习的概念和实践早在 20 世纪 60 年代就出现了，但是当代意义上的深度学习理论，公认是由 Hinton 于 2006 年提出的，其核心思想就是先利用竞争性非监督学习提取出有意义的特征，再用监督学习对这些特征进行识别，大大提高了学习效率。

在这类技术中，最著名的当属 Hinton 提出的深度信念网络。深度信念网络（DBN）是由一系列独立的受限玻尔兹曼机堆叠而成的，每个受限玻尔兹曼机（RBM）可以单独训练，并将其输出作为上层受限玻尔兹曼机的输入。受限玻尔兹曼机根据热力学理论，通过找到网络的能量最小值点来获得输入信号的特征表示，通过多层受限玻尔兹曼机，可以得到越来越抽象的特征，从而便于上层监督学习层的处理。将所有受限玻尔兹曼机训练完成并堆叠在一起之后，利用传统的误差反向传播算法进行微调，从而加快训练速度，减小对训练样本数量的需求，得到更高的识别精度，而且解决了之前困扰神经网络应用的一个难题，即对特定问题特征选取的问题。

深度信念网络在一系列应用上取得了很好的应用效果，例如 2006 年，Hinton 利用深度信念网络在没有经过训练样本变形的情况下，在 MNIST 手写数字测试集上的误差率达到 1.2% 左右。不仅在图像识别领域，在语音识别领域，深度信念网络在 TIMIT 测试集上的识别误差率也降低到了 26.7% 左右。基于深度信念网络的语义哈希索引技术，可以将相似的文档放在邻近的位置，比传统的邻域敏感哈希索引技术的效果有较大的提高。

与此同时，Bengio 于 2007 年提出，可以通过堆叠自动编码器，作为监督学习的预训练层，提高监督学习的效果。自动编码器就是先将输入信息输入到网络输入层，然后经过维度较低的隐藏层，最后输出到与输入信号维度相同的输出层。堆叠自动编码器的目标是使输出层的信号与输入层的输入信号尽可能一致，这样中间隐藏层的输出就可以视为输入信号的压缩编码。训练结束之后，可以将这个自动编码器的隐藏层作为输入，接入上层的自动编码器。为了提高自动编码器的性能，研究者们提出了各种改进方案，例如稀疏自动编码器、去噪自动编码器等模型，在实际应用中取得了较好的效果。

在 2006 年，出现了基于 GPU 的卷积神经网络实现，其比 CPU 上的卷积神经网络的速度快 4 倍左右，而一些基于 GPU 的算法，甚至比 CPU 的速度快 20 倍左右。正是这些早期

的实践，开创了当前主流深度学习算法依赖 GPU 的潮流。

在 2007 年，研究者在卷积神经网络中，将原来向下采样中使用的平均值池化技术改为最大池化技术，形成了所谓的 MPCNN，取得了非常好的应用效果。这种技术后来成为大量获奖项目的技术基础。

在 2007 年，研究者们提出了多层长短时记忆递归网络（LSTM RNN），这种网络输出层由连接序列分类（CTC）组成。对于序列标注，每个长短时记忆递归网络预测输入序列的标签，并将结果输入到上层长短时记忆递归网络。序列标注的误差，由顶层开始逐层向下反向传播，对连接权值进行修改。在语音数字识别中，这种网络的效果优于传统的隐马尔可夫模型，取得了较好的应用效果。

为了促进深度学习研究的发展，国际上有一系列深度学习算法竞赛，使一系列优秀的深度学习算法迅速流行起来。

在 2009 年的 ICDAR 手写字符识别竞赛中，利用了长短时记忆递归神经网络，输出层采用链式时序分类器算法，同时赢得了法语、阿拉伯语、波斯语识别竞赛，这是递归神经网络第一次赢得国际性神经网络大赛。

为了在监控视频中发现人物行为，研究者通过一个三维的卷积神经网络，结合支持向量机，组成了一个大型特征识别器，用来识别视频序列中感兴趣的内容。在 2009 年，这种架构的神经网络连续赢得了 3 个 TRECVID 国际竞赛。

2009 年也出现了 GPU 版本的深度信念网络实现，它比之前的 CPU 版本有了显著提升，同时有研究者提出，将卷积层引入深度信念网络，采用基于概率的最大池化模型，这在音频识别中得到了成功应用。

在 2010 年，在 MNIST 手写数字识别竞赛中，采用普通深层 BP 网络的误差率降到 0.35%。与其他流行的方法不同，它没有采用非监督学习的预训练网络。那么，一个古老的算法为什么能达到这么高的准确率呢？主要有以下两方面原因，首先是因为这种网络利用将训练样本变形的方式，产生大量训练样本，可以有效地避免过拟合问题；其次是这种网络采用了 GPU 实现，比之前的算法快大约 50 倍。由此可见，一些经典算法，在强大的运算能力帮助下，可以产生惊人的效果。这甚至从一个侧面说明，现代计算能力的进步，相比算法进步而言，更能推动深度学习的进展。

从 2011 年开始，基于 GPU 实现的卷积层，在最大池化向下采样，多层累加之后接入全连接层，通过 BP 算法进行权值调整学习，成为此后国际深度学习算法竞赛获奖项目的主流技术。此后，研究者们将多个这种类型的独立网络组合成卷积神经网络集群，系统最终识别结果是由各组成的卷积神经网络的判断结果在概率论的基础上进行综合汇总得出的。在 2011 年交通标志识别竞赛中，采用这个架构的神经网络的误差率降到了 0.56%，优于人类识别效果的两倍，比其他深度学习技术好 3 倍以上。由此可以看出这种网络架构的强大威力。采用这种架构的网络，在 MNIST 测试集上的误差率可以降低到 0.2% 左右，同样优于人类的识别能力。

在图像识别竞赛中，ImageNet 比赛的影响力无疑是最大的。在 ImageNet 竞赛中，深度学习算法识别、物体检测或分割分辨率为 256×256 的图片，其中用于训练的图片有 1200 万

张左右，用于验证集的图片有 500 万张，用于测试和评价的图片有 100 万张，所有这些图片属于 1000 个类别。在 ImageNet 的所有竞赛项目中，图像中的物体识别是最令人关注的项目。因为物体识别可以用于图片搜索、人体组织医学图像诊断，例如识别出癌组织，但是图像中的物体识别也是非常困难的。在 2012 年，基于 GPU 的卷积神经网络集群，赢得了乳腺癌组织图像识别的比赛。这个结果具有重要的意义，全美国超过 10% 的 GDP 用于医疗健康产业，其中很大一部分用于医疗诊断，而最主要的应用就是医学图像诊断，目前这部分工作完全需要高水平专家来完成，如果可以部分地实现自动化，这将会极大地降低医疗成本，同时使更多的大众可以享受专家级的诊断技术服务。

也是在 2012 年，在图像分割领域，基于 GPU 的最大池化卷积神经网络集群表现出色。欧洲和美国的由电子显微镜获取的脑图像数据，需要重建出 3D 神经元和树突的模型，如果这部分由人工来对图像进行注解，需要数天甚至数周的时间才能完成，非常需要自动化。

1.2 深度学习现状

1.2.1 传统神经网络困境

传统神经网络或机器学习算法虽然可以成功解决很多问题，但是在语音识别、图像识别、图像标注等领域却遇到了巨大的问题。传统机器学习算法的一个主要问题就是泛化问题，泛化能力指训练好的模型遇到新的样本时做出正确判断的能力。在低维的情况下，这个问题并不严重，但是一旦遇到高维问题，泛化问题的解决难度将呈现指数级增长的趋势，而这一问题只能通过深度学习才能解决。在本节中，我们将讨论传统机器学习算法在高维问题中遇到的困境，以及深度学习算法给出的解决方案。

传统机器学习遇到的第一个问题就是所谓的维度诅咒问题。在高维甚至超高维问题中，传统机器学习遇到的问题首先是由于维度增加，造成模型参数指数级增长的问题，同时造成在参数空间中找到最优解的难度和计算量均呈指数级的增长趋势，因此传统机器学习算法在高维问题中极少取得成功。

假设我们研究 3 个问题，第一个问题的维度为 10，第二个问题的维度为 100，第三个问题的维度为 1000。

对于第一个问题，由于问题的维度是 10，每个维度上有 100 个不同的值，则将有 1000 种组合。要想处理这种情况，至少需要 1000 个以上的训练样本，而且它们必须均匀分布在每个不同的组合中。这样当一个需要识别的新样本出现时，尽管我们用的具体算法可能不同，但是其原理都是找到与这个样本最像的样本，再根据与其最像的样本来预测新样本的类别。1000 个训练样本的需求还是相对容易满足的，所以传统的机器学习算法还是有可能成功应用于这类问题的。

对于第二个问题，由于有 100 个维度，如果每个维度上也有 100 个不同的值，则有 10000

种不同的组合。要想处理这种情况，我们至少需要 10 000 个以上的训练样本，而且它们必须均匀分布在每个不同的组合中。这样大的样本量，而且要求在每种组合中都有足够多的训练样本供机器学习算法来学习，就非常难以达到了。因为在通常情况下，我们即使有大量的样本，也更有可能集中在其中的常用类别中，所以很可能无法满足要求。在这种情况下，传统的机器学习算法就很难应用在这类问题上了。

至于第三个问题，由于有 1000 个维度，如果每个维度上还是有 100 个不同的值，则将有 100 000 种不同的组合。要想处理这种情况，至少需要 100 000 个以上的训练样本，而且必须在每个组合中都具有训练样本。这将是一个非常苛刻的要求，在实际应用中基本不可能满足。因为在实际应用中，即使我们有海量的训练样本，这些样本更有可能集中在少数几个类别中，而其他类别中没有相应的样本，这样我们就无法进行有效的学习。这就是传统机器学习算法在现代高维度问题中难以应用的重要原因。

传统机器学习算法面临的第二大问题是局部一致性和平滑性假设。为了提高泛化能力，机器学习算法需要一些先验知识作为指导，告诉深度学习算法要学习的函数具有哪些特性。例如，我们认为小的权值有利于提高泛化能力，因此通常有 L2（权值衰减）调整项。当然有一些先验知识直接作用于需要学习的函数，而间接作用于参数。

最常用的先验知识是局部一致性或光滑性，即我们认为目标假设函数在局部是光滑的，一个小的值域上的改变不会引起函数值产生巨大的变化。

正是由于传统的机器学习算法严重依赖于局部光滑性假设，所以无法处理复杂的问题，而深度学习算法通过引入额外的先验知识，成功地解决了这一问题。下面先来看一下为什么只依赖于局部光滑性先验知识会有问题，然后再来看一下深度学习怎样解决这一问题。

我们可以将局部光滑性表示为：

$$f^*(x) \approx f^*(x+\epsilon)$$

在上式中， ϵ 是一个非常小的值，加上 ϵ 后的计算结果与未加之前的计算结果基本相等。根据这一假设，对于一个新样本点，我们只需找到与其足够相邻的点，通过某种求平均和內插法，可以很好地根据训练样本推断新样本的属性。对这个性质应用得最极端的例子是 K 最小近邻算法。

这种先验知识很简单也很好用，但缺点就是如果某个区域没有训练样本，基于这个假设的学习算法就不能应用了。那么有没有办法解决这类问题呢？例如我们要研究的问题是一个国际象棋棋盘，其由黑白相间的方格组成，在局部光滑性假设下，如果某个方格内有训练样本并且在居中位置的话，我们就可以很准确地判断样本是黑色的还是白色的。但是对于没有训练样本的方格，当新样本在这些方格中时，我们无法做出正确预测。如果我们这时有先验知识，方格是黑白相间的，那么即使没有训练样本的方格也可以很好地做出预测。深度学习算法就是提出了类似的先验知识，所以可以解决很多传统机器学习算法无法解决的问题。

当然，在实际的深度学习工作中，我们不可能使用像上面的先验知识，因为实际问题中很难有像上例国际象棋棋盘这么规则的问题，另外，为了我们的深度学习算法更具通用性，也不倾向于使用过于特例化的先验知识。

深度学习最常见的一个先验知识就是，认为复杂问题是由具有层次结构的特征组成的，这些特征是可以分步来学习的，深度学习网络就是将上述每一步用深度学习网络的层来表示。我们可以证明，同样一个问题，短而浅的网络所需的参数要远远大于深而窄的深度学习网络。网络参数减少，学习难度自然会降低，这就与我们的日常生活常识相一致。例如，有一道非常难的数学题，通过分步求解就比较简单，但是如果让我们直接说出答案则非常困难。

深度学习中另外一个非常重要的概念就是流形学习。流形是一个互相连接的区域。流形学习是最近比较热的一个机器学习领域，数学模型和概念描述比较抽象难懂，所以这里就不讨论了。我们这里仅介绍一些实际例子，让大家对这个新兴的领域有一个感性的认识。

我们先将一张纸团成一团，然后再将其展开平放在阳光下，这时纸面上会有很多复杂的褶皱，形状非常不规则。如果我们需要学习的函数就是这样一张纸，那么将非常复杂。但是如果我们的研究纸在阳光下的影子，那么纸就变成了二维模型，难度就大幅降低了。

再以一个球面为例，我们所研究的样本点都在一个球心为 (x_0, y_0) 、半径为 r 的球面上，而不是在普通三维空间上。那么，对于三维球面来说，可以用球面方程来表示：

$$x = x_0 + r \sin \theta \cdot \cos \varphi$$

$$y = y_0 + r \sin \theta \cdot \sin \varphi$$

$$z = z_0 + r \cos \theta$$

式中， $\theta \in [0, \pi]$ ， $\varphi \in [0, 2\pi]$ 。

如果我们的应用分类问题假设 $\theta \in [0, \frac{\pi}{2}]$ ， $\varphi \in [0, \pi]$ 是一类，其他的是另一类。那么这两个类别在三维空间上来看就是一个复杂的球面分割问题。然而如果把这些点根据上面的方程组求解出对应的 θ 、 φ 值，那么我们不仅成功地将问题从三维变为二维，更重要的是，在 θ 、 φ 二维空间上，类别的分割界面将是一个矩形，这无疑使问题得到了极大的简化。由此可见，如果我们能够找到流形学习背后的规律，就能化繁为简，使很多复杂问题迎刃而解。

上面讨论的问题过于简单了，下面来看一个比较实际的例子。以 MNIST 手写数字识别为例，原始图像为 28×28 的黑白图片，输入到机器学习算法时，就变成了 784 维的问题，但是并不是说 784 维上的每个点我们都关心，实际上我们只关心 6 万多个点，这些点对应于 0~9 这 10 个数字的手写字体。这个例子实际上与上面的球面例子类似，我们直接在原始的 784 维空间上求解将十分困难，如果能找到类似球面方程的形式，那么问题就变得简单多了。但是 MNIST 手写数字识别很难找出类似的方法，如果通过卷积神经网络还是可以找出图片所对应的特征，从而在这些特征空间中学习，大大提高学习效率。

1.2.2 深度多层感知器

深度前馈网络一度被认为是一种低效的网络，在非常长的时间里不被人们所重视。因

为有定理证明，对于任意复杂的函数，都可以通过选择合适的隐藏层神经元数，达到以任意精度逼近原函数的网络，这就是著名的万能逼近理论。以此理论为指导，10 年之前，几乎所有成功的前馈网络基本上都是 3 层结构，只有 1 个隐藏层，不同的是隐藏层神经元数量会有比较大的差异，但卷积神经网络是一个特例。因为在这种网络中，采用了局部接收域、权值共享、最大池化等技术，成功地减少了训练网络所需要的参数，在图像应用领域取得了巨大成功。但是这毕竟仅是一个特例而已，当时主流的神经网络应用还是以 3 层前向神经网络为主。

直到 2006 年，Hinton 教授首先采用受限玻尔兹曼机堆叠形成深度信念网络，并在 MNIST 手写数字识别中取得成功，这才第一次证明深度前馈网络只要采用合理的技术手段也是一种非常强大的机器学习技术。在 2014 年左右，完全基于 BP 算法的前向神经网络没有任何预学习过程，在 MNIST 手写数字识别竞赛中获得冠军，人们终于认识到深度前馈网络确实是一种非常强大的学习技术。

深度前馈网络的核心技术误差反向传播算法，已经至少三四十年没有发生变化了，但是为什么近年深度前馈网络会重新流行起来呢？这主要是由以下三个方面的因素共同决定的。

第一方面无疑是计算技术的发展。近年来，随着计算能力的提高，尤其是 GPU 在深度学习领域的应用，人们拥有的计算能力是三四十年前难以想象的。另外，随着 Web 2.0 等技术的发展，出现了很多巨型的数据集。我们知道，前馈网络的参数非常多，因此需要大量的训练样本数据，UGC 的出现正好提供了这样的数据，因此深度学习算法首先在这方面获益也就不足为奇了。

第二方面来自算法层面的改进。之前人们一直使用平均平方误差作为前馈神经网络的代价函数，但是由于我们采用 Sigmoid 或双曲正切作为神经元的激活函数，而这些函数在过大或过小的数值上都趋近于饱和，其导数将非常小，因此会使学习过程变得相当困难。近年来，人们普遍采用交叉熵函数，将负对数似然函数作为代价函数，因为在负对数似然函数中会取对数，其正好可以中和神经元激活函数中的指数函数所带来的饱和问题，使学习过程变得相对容易。

第三方面来自神经元的选择方面。在此之前，人们通常倾向于选择采用 Sigmoid 函数作为神经元激活函数，或者因为效率问题，采用双曲正切函数作为神经元激活函数。因为人们认为 Sigmoid 函数在 $(-\infty, +\infty)$ 区间连续且可导，并且值域为 $(0,1)$ ，可以代表事件的发生概率，是理想的神经元激活函数。但是后来人们发现了 ReLU 函数，尽管其在 $x=0$ 点不可导，但是实际效率却要好很多。由于 ReLU 神经元高效、简单的特性，目前其已经成为前馈神经网络隐藏层神经元的默认选择。另外，研究人员也发现，对于小数据集，ReLU 神经元比其他神经元有更好的性能，并且优势相当明显。神经科学方面的研究成果也支持 ReLU 神经元是一种更接近于生物神经元的模型，因为神经科学方面的研究发现，生物神经元具有以下特性：① 对于某些输入信号，神经元没有任何反应；② 对于某些输入信号，神经元的输出与输入信号成正比；③ 在大多数时间，神经元工作在非活跃状态。前两个特性与 ReLU 神经元的特性完全一致，这也从一个侧面证明了 ReLU 神经元的合理性。

当前，人们普遍认为，深度前馈神经网络是一种非常强大的神经网络，还有非常大的发展潜力。研究人员正在将深度前馈网络作为工具，来发展变分自动编码器（Variational Autoencoder）和生成式对抗网络（Generative Adversarial Network, GAN）。在深度学习兴起之初，深度前馈神经网络被认为需要非监督学习算法来协助才行，而现在深度前馈神经网络更经常的角色是协助非监督学习来更好地完成任务。可以预见，随着计算能力的提高和数据集的增大，深度前馈神经网络还会取得更大的成功。

1.2.3 深度卷积神经网络

卷积神经网络在深度学习历史上具有重要地位。卷积神经网络是通过研究脑神经系统受到启发而研制的系统，研究人员受到生物视觉神经系统具有接收域的启发，设计了层间具有稀疏连接的神经网络系统，同时利用最大池化技术实现了一定的移动不变性，通过层叠接收域体系，形成深度神经网络，并最终在图像处理领域取得了巨大成功。现在卷积神经网络不仅在图像处理领域，而且在视频处理、fMRI 等医学图像处理、语音识别等领域也取得了巨大的成功。

卷积神经网络也是第一个成功的深度神经网络。在现代深度学习算法复兴之前，人们普遍认为深度网络不具有实践价值，而选用浅而宽的神经网络来解决实际问题。但是以 3×3 为接收域的卷积神经网络可以达到几十层，同时还可以具有很好的泛化效果。研究人员基本形成共识，在卷积神经网络中，窄而深的网络具有更好的性能。卷积神经网络为后来的深度学习算法复兴提供了宝贵的实践经验。

卷积神经网络同时是目前商业应用上较成功的神经网络。早在 20 世纪 90 年代，AT&T 的研究人员就用卷积神经网络成功解决了账单读取问题，到 20 世纪 90 年代末，这个系统在 NEC 上可以读取 10% 左右的账单。微软也在 2000 年前后开始应用卷积神经网络，并且在光学字符识别和手写识别方面取得了巨大的成功。从 2012 年开始，ImageNet 竞赛的冠军基本被卷积神经网络及其变形所控制，在 MNIST 手写数字识别竞赛中，目前大幅超过人类识别水平的记录，也是由卷积神经网络来保持的。

综上所述，卷积神经网络不仅是目前较成功的神经网络技术，同时也是未来具有广阔发展前景的网络。笔者认为，今后卷积神经网络的研究重点，除理论上解决为什么卷积神经网络比全连接网络性能要好之外，还需要研究有哪些在实际中取得成功的卷积层卷积核、它们分别适用于哪些特定情况，以及怎样动态选择这些卷积核。例如，我们能够确定哪些卷积核对草原的识别效果最好，哪些卷积核对斑马的识别效果最好，如果当前是一幅草原上有几匹斑马的图片，那么我们应该可以动态组成一个卷积神经网络。在周围采用对草原识别效果好的卷积核，而在中间使用对斑马识别效果好的卷积核，这样可以得到更好的特征，便于后续进行基于特征空间的模式识别学习。

1.2.4 深度递归神经网络

递归神经网络是一种允许信号回传的神经网络，普通递归神经网络由一个 3 层网络组成，中间的隐藏层不仅将输出值传送给输出层，还将该值传输给自己，这样一个输入信号的作用就不仅仅是其输入的一瞬间，而是在其后很长时间持续起作用。为了控制输入信号的作用范围，研究人员在 20 世纪 90 年代后期引入了长短时记忆网络，通过引入输入门、遗忘门、输出门来控制。当人们不希望新的输入信号影响现有输入信号时，就关闭输入门。如果旧的输入信号处理完成，需要处理新的信号时，遗忘门打开，将原来的输入信号影响消除，再打开输出门，就可以在输出层得到有用的信息。由于可以对任意长度的序列进行标注，递归神经网络在语音识别、手写文字识别，甚至图像处理领域都有非常成功的应用。

递归神经网络也是一种结构最为灵活的网络，为了处理手写文字识别，如果知道下面几个字母，可以根据单词拼写更好地识别当前字符。研究人员提出了双向递归神经网络，其由两个递归神经网络构成，一个是从头到尾读序列，另一个是从尾到头读序列，这在手写文字识别中取得了巨大的成功。为了处理序列的开始和结束不具有明确的规则的问题，研究人员提出将普通输出层替换为 CTC 层，这在语音识别领域同样取得了巨大的成功，其比之前一直采用的隐马可夫模型的准确率大幅提高。

从传统意义上讲，递归神经网络的应用领域主要集中在语音识别等序列标注领域，但是最近研究人员提出多维递归神经网络（MD-RNN），用于处理图像识别任务，其虽然运算量较大，但是泛化效果要优于其他神经网络。与卷积神经网络级联的接收域处理图像信息相似，研究人员还提出了多维级联递归神经网络，并成功用于图像识别领域，且取得了较好的效果。

递归神经网络是结构最灵活的神经网络，同时也是功能最强大的神经网络，研究人员一直在提出各种改进模型，以提高递归神经网络的性能，其中最具代表性的当属神经图灵机（Neural Turing Machine, NTM）。

智能需要知识，知识可以通过学习来获取。正是基于这个理念，人们建立了各种大型深度学习架构。但是知识分为不同的类别，有些知识很难用语言描述，比如如何走路、猫和狗有什么区别等。而另外一些知识，则可以用语言来明确表示，例如猫是一种动物、这些知识可以使用知识图谱来进行描述等。

神经网络在存储隐式知识方面表现很好，却很难表示显式知识。在神经网络训练中，最典型的是使用随机梯度下降算法，要把一个输入信号对应的模式保存到网络中，需要多次迭代才能完成，而且即使如此，神经网络对这个知识的记忆也是不稳定的，完全可能因为学习其他样本而将这个样本忘记，这也是我们在训练样本集上会有误差的原因。

针对这种情况，Graves 在 2014 年提出新的假设，认为神经网络之所以不能很好地记忆知识，是因为神经网络缺少显式的工作记忆。人类在解决某项任务时，通常会先从工作记忆中查找相关知识，然后根据这些知识来决定怎样完成一项任务的问题。

为了解决这个问题，研究人员提出了内存网络的概念，其包括一系列记忆单元，可以通过一种特定的寻址机制来获取记忆单元的内容。最初，这些网络需要人工监督才能将信

息存入记忆单元。2014 年 Graves 提出的神经图灵机，通过采用一种软寻址机制，也就是软注意力机制，可以将任意内容从记忆单元读出或写入，而不需要人工的监督信号，从而使采用梯度下降算法进行学习成为可能。

通过向递归神经网络引入注意力机制和工作记忆单元，并且提供基于内容或空间的寻址方式，类似神经图灵机的网络表现出了巨大的应用前景，这是一个非常有前途的研究方向。

1.3 深度学习研究前瞻

在 1.2 节中，我们向读者介绍了当前比较成熟的深度学习网络技术。当然，深度学习到目前为止还在高速发展之中，包括 1.2 节介绍的神经网络技术，其自身还在不断发展完善之中，这里指的成熟是指其已经有了比较成功的商业应用。

在本节中，我们将介绍深度学习的研究前沿，这些技术目前还处于萌芽状态，虽然还没有特别成功的商业应用，但是由于其技术上的创新性，有望在未来的商业应用中取得巨大的成功。我们学习深度学习算法，对于这些内容也一定要有适当的了解，也许在两三年之后，这里所讲的技术就会像卷积神经网络一样取得重大的商业成功。

我们之前介绍的深度学习技术，都可以归纳为监督学习的类别。这类问题本质上是大量经过标注的数据来训练我们的网络模型，再拿训练好的模型来解决实际中的问题。对于这个任务，之前介绍的技术已经做得非常成功了。但是其中还有一个问题，那就是这些技术都需要人工标注的数据，而且需求量非常大。但是在很多情况下，我们无法获取这些标注数据，因此想要应用前面的技术就非常困难了。

本节我们主要讨论怎样生成新的样本，或者决定哪些样本之间更相像一些，找到缺失的特征值（通常因为噪声造成特征值的缺失），或者进行迁移学习。通过采用这些技术，我们可以极大地减小对人工标注数据的依赖，使深度学习技术可以应用到更加广泛的领域。要解决这些问题，就必须借助非监督学习或半监督学习。

造成非监督学习困难的主要原因是：研究问题的高维度。研究问题的高维度会带来两个方面的挑战：统计学挑战和计算挑战。统计学挑战主要是泛化问题，因为维度高，必然使参数组合状态呈指数级的增长，而我们的训练样本往往是严重不足的，无法在所有参数组合中都有样本存在，这就造成了学习的困难。计算挑战主要是指由于维度升高，所需计算量也呈指数级增长，这使得目前的计算技术无法在合理的时间内得到希望的结果。当采用概率模型时，在高维情况下，我们同样会遇到两大类问题：不可控推理和不可控归一因子问题。不可控推理是指假设我们有一个模型，可以得到随机变量 A 、 B 、 C 的联合概率分布，例如在给定随机变量 B 的条件下，要求随机变量 A 的值。为了计算这个条件概率，我们需要对随机变量 C 的所有值进行求和，同时还必须计算随机变量 A 和 C 的归一化因子，通常这些计算的运算量都非常大，常常因为过大而无法进行计算。不可控归一因子问题是指，在很多模型中需要计算所有的概率分布，例如受限玻尔兹曼机网络中对于可见层的计

算。假设可见层神经元数为二值神经元（仅能取 0、1 两个值），共有 m 个神经元，那么其所有的状态为 2^m ，这时即使可见层只包含几十个神经元，直接计算归一化因子的计算量也是一个天文数字。

为了解决上述提到的问题，研究人员一方面提出了近似方法，因为直接计算可能运算量非常巨大，但是通过适当的近似化简之后，模型得到了简化，我们就可以通过有限步骤的计算得出希望的结果。

另一种解决这些问题的思路是，完全避免计算这些量，采用新方法和模型来解决这些问题，这就是所谓的生成式模型。

最近，研究人员倾向于将生成式模型与监督学习结合起来，形成生成式对抗网络，这在实践中取得了较好的效果，是一种特别具有发展潜力的研究方向。在生成式对抗网络中，我们首先训练生成式模型，然后通过采样生成新的训练样本，交给模式识别网络进行识别，通过反复这一过程，使生成式网络产生的样本越来越像实际的样本，而模式识别网络越来越能正确识别实际中出现的样本。通过这种对抗机制的竞争关系，使生成式网络和模式识别网络共同进步，这就像在大自然中，捕食者和被捕食者相互竞争，最终造就了强大的捕食者和灵活机动的被捕食者一样。

1.3.1 自动编码器

自动编码器是一种典型的非监督学习网络，网络由 3 层组成，从下向上分别为输入层、中间层和输出层。先将输入信号加载到输入层，输入层将信号传输到中间层，中间层再将信号传输到输出层。通常输入层神经元数与输入信号的维度相同，输出层神经元数与输入信号的维度也相同，而中间层比输入层、输出层神经元数要少。当我们将输入信号加载到输入层之后，取出输出层的输出信号，检查输出层输出信号与输入信号的差异，网络学习的目的就是使输出层输出信号与输入信号间的差异最小化。而这一过程所采用的学习方法，完全可以采用监督学习的方法，把输入信号视为已经标注好的结果数据。自动编码器经过训练，在输出层可以得到任意逼近输入信号的输出值。这时中间层可以被视为输入信号的另一种表现形式，输入信号从输入层传输到中间层可以被视为对输入信号进行了某种编码，得到其在一个新的特征空间中的表示，而从中间层到输出层则可以被视为将信号从新的特征空间表示形式转回原来的空间表示形式，而在原来空间的表现形式和在新的特征空间的表现形式具有等效性。由于中间层的神经元数比输入层、输出层要少，因此新的特征空间的维度会比原始输入信号的维度低，这样就起到了降维的作用。我们知道，高维通常是机器学习不能高效解决实际问题的一个主要原因，自动编码器这种降维能力，正是我们所需要的。

以上所讨论的是普通自动编码器，但是普通自动编码器在实际应用中会存在比较大的问题。因为普通自动编码器只能对已知输入信号进行编解码，对于未知输入信号，如果它与已知输入信号差别较大，则自动编码器的效果就会比较差，不具有实际使用价值。为了

解决这个问题，研究人员提出了两种解决方案：去噪自动编码器、稀疏自动编码器。

去噪自动编码器是指通过向输入信号加入随机噪声，将加入噪声后的输入信号加载到输入层，经过中间层传输到输出层，并且试图使输出层的输出信号与原来未加入噪声的输入信号相同。去噪自动编码器试图从噪声污染的信号中恢复出原始信号，通过这种方式来提高遇到新样本时编码的性能。

稀疏自动编码器与普通自动编码器略有不同，在稀疏自动编码器中，中间层的神经元数不一定比输入层、输出层的少，而且可能比输入层多很多，但是通过特定的学习算法，不同的输入信号会激活少数特定的中间层神经元，而绝大多数其他神经元处于非激活状态，即具有稀疏性。通过稀疏性编码可以达到降维的目的，也可以提高网络遇到新样本时的泛化能力。

自动编码器可以用于信号压缩领域，也可以用于高维问题的降维处理。通常自动编码器并不单独使用，而是先将自动编码器进行堆叠，即将低一层自动编码器的中间层作为上一层自动编码器的输入层，以此类推，并在顶层加入逻辑回归层，形成多层深度网络，然后用类似多层感知器（MLP）的算法进行调优。这种通过非监督学习来辅助监督学习的方式，是深度学习算法复兴初期的主流方法。

1.3.2 深度信念网络

当非监督学习在高维问题中遇到难题时，我们实际上可以绕开统计学和计算方面的问题，采用生成式学习的方式。下面就来讨论一下生成式网络。

受限玻尔兹曼机是一种典型的生成式网络，同时也是图形模型在深度学习方面的典型应用。受限玻尔兹曼机是一种无向图，同时也是一个能量模型。通过定义的能量函数，可以了解系统每个状态下对应的能量值，受限玻尔兹曼机的学习任务就是找到能量最低时所对应的状态。

受限玻尔兹曼机并不是一种深度学习技术，其主要是级联在一起，再在最上面叠加一层用于分类的逻辑回归模型，形成所谓的深度信念网络。深度信念网络的训练由逐层预训练和调优网络训练两阶段来完成。

在逐层预训练阶段，我们先从下向上单独训练单个受限玻尔兹曼机，训练完成之后，将该受限玻尔兹曼机的隐藏层作为上一层受限玻尔兹曼机网络可见层的输入，以此类推，直到最后一个受限玻尔兹曼机训练完成为止。

在训练好的受限玻尔兹曼机之上叠加分类网络，通常是线性回归层，形成一个类似多层感知器的网络，采用多层感知器模型的训练方法对网络进行训练，直到达到满意的效果为止。

这种采用非监督学习协助监督学习的方法，在深度学习系统刚刚兴起时是一种主流的深度学习训练方法。

受限玻尔兹曼机同时也是一种生成式网络，在第 10 章读者将看到，如果我们对训练好

的受限玻尔兹曼机的可见层进行采样，就可以得到与输入信号非常像的生成信号。生成式网络的这种特性，在训练样本缺乏的情况下，可以通过对可见层进行采样生成训练样本。在对这些样本进行人工标记之后，还可以反过来对网络进行训练，以达到更好的性能。

1.3.3 生成式网络最新进展

在具体介绍生成式对抗网络之前，我们先来讨论一下生成式网络模型。

我们知道机器学习方法可以分为生成方法（Generative Approach）和判别方法（Discriminative Approach），所对应的模型分别被称为生成式模型（Generative Model）和判别式模型（Discriminative Model）。

生成方法通过观测数据学习样本与标签的联合概率分布 $P(X,Y)$ ，训练好的模型能够生成符合样本分布的新数据，可以用于监督学习和非监督学习。在监督学习任务中，根据贝叶斯公式由联合概率分布 $P(X,Y)$ 求出条件概率分布 $P(Y|X)$ ，从而得到预测的模型，典型的模型有朴素贝叶斯模型、混合高斯模型和隐马可夫模型等。非监督生成模型通过学习真实数据的本质特征，刻画出样本数据的分布特征，生成与训练样本相似的新数据。生成模型的参数远远小于训练数据的量，因此模型能够发现并有效内化数据的本质，从而可以生成这些数据。生成式模型在非监督深度学习方面占据主要位置，可以用于在没有目标类标签信息的情况下捕捉观测到或可见数据的高阶相关性。深度生成模型可以通过从网络中采样来有效生成样本，例如受限玻尔兹曼机、深度信念网络（Deep Belief Network, DBN）、深度玻尔兹曼机（Deep Boltzmann Machine, DBM）和广义去噪自编码器（Generalized Denoising Autoencoder）。根据 OpenAI 的研究，近两年来流行的生成式模型主要分为以下三种。

1. 生成式对抗网络（Generative Adversarial Network, GAN）

GAN 启发自博弈论中的二人零和博弈，由 Goodfellow 在 2014 年 NIPS 会议论文中开创性地提出，包含一个生成模型和一个判别模型。生成模型捕捉样本数据的分布，判别模型是一个二元分类器，判别输入是真实数据还是生成的样本。模型的优化过程是一个“二元极小极大博弈（Minimax Two-Player Game）”问题，训练时固定一个模型，更新另一个模型的参数，交替迭代，使对方的错误最大化，最终估测出样本数据的分布。

2. 变分自编码器（Variational Autoencoder, VAE）

在概率图模型（Probabilistic Graphical Model）的框架中，对问题进行形式化——在数据的对数似然上最大化下限（Lower Bound）。

3. 自回归模型（Autoregressive Model）

PixelRNN 这样的自回归模型通过之前给定的像素（左侧或上部），以及通过对每个单个像素的条件分布建模来训练网络。这类似于将图像的像素插入 char-rnn 中，但该 RNN 在图像的水平 and 垂直方向上同时运行，而不只是字符的 1D 序列。

在所有这些最新的生成式网络中，目前风头最劲的当属生成式对抗网络了。简单来说，在一个对抗网络中，判别器的输出就是：遇到真实图片，输出 1；遇到生成图片，输出 0。判别器想要做好这项工作，因此它会优化自身，防止被生成器欺骗。反过来，生成器也在优化自己，它想生成非常真实的图像，尽可能地迷惑判别器，让其难辨真伪。最后，生成器开始生成非常真实的图片：无论图片是生成器生成的还是真实的，大多数情况下，判别器的正确率都是恒定的。

这种对抗训练过程与传统神经网络存在一个重要区别。一个神经网络需要有一个成本函数，评估网络性能如何。这个函数构成了神经网络学习内容和学习情况的基础。传统神经网络需要一个人类科学家精心打造的成本函数，但是对于生成式模型这样复杂的过程来说，构建一个好的成本函数绝非易事，这就是对抗性网络的闪光之处。对抗网络可以学习自己的成本函数——自己那套复杂的对错规则——无须精心设计和构建一个成本函数。

1.4 深度学习框架比较

在目前的情况下，开展深度学习研究和应用，通常不需要从头开始重新实现深度学习算法，可以在成熟的开源深度学习框架下，高效地完成我们的任务。在本节中，我们将向大家简单介绍当前主流的开源框架、它们各自的优缺点，以及选择过程中需要注意的问题。

1.4.1 TensorFlow

TensorFlow 是谷歌基于 DistBelief 研发的第二代人工智能学习系统，其命名来源于本身的运行原理。Tensor（张量）意味着 n 维数组，Flow（流）意味着基于数据流图的计算，TensorFlow 为张量从图像的一端流动到另一端的计算过程。TensorFlow 是将复杂的数据结构传输至人工智能神经网络中进行分析 and 处理的系统。

TensorFlow 表达了高层次的机器学习计算，大幅简化了第一代系统，并且具备更好的灵活性和可延展性。TensorFlow 的一大亮点是支持异构设备分布式计算，它能够在各个平台上自动运行模型，从单个 CPU/GPU 到成百上千个 GPU 卡组成的分布式系统。从目前的文档看，TensorFlow 支持 CNN、RNN 和 LSTM 算法，拥有 C++/Python 编程接口，这些都是目前在 Image、Speech 和 NLP 上最流行的深度神经网络模型。

TensorFlow 的数据结构是 tensor，它相当于 n 维的 array 或 list，与 MXNet 类似，都采用以 Python 调用的形式展现出来。某个定义好的 tensor 的数据类型是不变的，但是维度可以动态改变。用 tensor rank 和 tensor shape 来表示它的维度（例如 rank 为 2 可以被看成矩阵，rank 为 1 可以被看成向量）。tensor 特别的地方在于，在 TensorFlow 构成的网络中，tensor 是唯一能够传递的类型，而类似于 array、list 这些类型则不能被当成输入。

TensorFlow 的网络实现方式选择的是符号计算方式，它的程序分为计算构造阶段和执行阶段。计算构造阶段是构造出计算图（Computation Graph），计算图就是包含一系列符号操作 Operation 和 Tensor 数据对象的流程图，和 MXNet 的 symbol 类似，它定义好了如何进行计算（加、减、乘、除等）、数据通过不同计算的顺序（也就是 Flow，数据在符号操作之间流动的意思）。但是暂时并不读取输入来计算获得输出，而是由后面的执行阶段启动 session 的 run 来执行已经定义好的 graph。这样的方式和 MXNet 很相似，应该都借鉴了 Theano 的想法。其中 TensorFlow 还引入了 Variable 类型，它不像 MXNet 的 Variable 属于 symbol（TensorFlow 的 operation 类似于 MXNet 的 symbol），而是一个单独的类型，主要作用是存储网络权重参数，从而能够在运行过程中动态改变。TensorFlow 将每一个操作抽象成一个符号 operation，它能够读取 0 或多个 Tensor 对象作为输入（输出），操作内容包括基本的数学运算、支持 reduce 和 segment（tensor 中的部分）进行运算。

TensorFlow 的优点如下。

- （1）TensorFlow 功能很齐全，能够搭建的网络很丰富，而不像 Caffe 仅仅局限在 CNN。
- （2）TensorFlow 的深度学习部分，能够在一个模型中堆积许多不同的模型和变换，能够在一个模型中方便地处理文本、图片、规则分类及连续变量，同时实现多目标和计算多损失函数的工作。
- （3）TensorFlow 的管道部分能够将数据处理和机器学习放在一个框架中，由 TensorFlow 来指定计算图运行方向。

TensorFlow 是一个理想的 RNN API 和实现，TensorFlow 使用了向量运算的符号图方法，使得新网络的指定变得相当容易。但 TensorFlow 并不支持双向 RNN 和 3D 卷积神经网络，同时公共开源版本的图定义也不支持循环和条件控制，这使得 RNN 的实现并不理想，因为必须要使用 Python 循环且无法进行图编译优化。与此相反，Theano 则对这些需求提供了较好的支持。

1.4.2 Theano

由于本书中的主要例子均使用 Theano 框架，因此在这里我们就不对 Theano 进行详细的介绍了，只是简单地列举一下它的优缺点。总体来说，Theano 是一个 Python 库，用来定义、优化和计算数学表达式，以及高效解决多维数组的计算问题。

Theano 的优点如下。

- ❑ 集成 NumPy 库，使用 numpy.ndarray 来表示多维信号。
- ❑ 使用 GPU 加速计算，比 CPU 快 140 倍（只针对 32 位 float 类型）。
- ❑ 有效的符号微分，计算一元或多元函数的导数。
- ❑ 速度和稳定性优化，比如能计算很小的 x 的函数 $\log(1+x)$ 的值。
- ❑ 动态地生成 C 代码，更快地计算。
- ❑ 广泛地单元测试和自我验证，检测和诊断多种错误。

- ❑ 灵活性好。

Theano 的缺点如下。

- ❑ scan 中参数传递机制非常难用，immutable 机制导致函数编译的时间过长。
- ❑ Theano 定义 function 时缺乏灵活的多态机制。
- ❑ 调试困难（这也许是 Theano 最大的缺点）。

1.4.3 Torch

核心的计算单元使用 C 语言或 CUDA 技术，同时做了很好的优化。在此基础上，使用 Lua 构建了常见的模型。另外，Torch 7 构建的是一个生态系统，安装新的模型实现模块只需要 `luarocks install package`，比如 `luarocks install rnn`。之后就可以非常简单地使用 RNN 模型了。

核心特征总结如下。

- (1) 一个强大的 n 维数组。
- (2) 很多实现索引、切片、移调的例程。
- (3) 高效的基于 LuaJIT 的接口。
- (4) 线性代数例程。
- (5) 神经网络，基于能量的模型。
- (6) 数值优化例程。
- (7) 快速高效的 GPU 支持。
- (8) 可嵌入、移植到 iOS、Android 和 FPGA 的后台。

Torch 的优点如下。

- (1) 构建模型简单，一层层搭积木即可。
- (2) 高度模块化，一层就是一个模块，写新模块也方便，套用接口就行，用 `tensor` 运算时不必写 CUDA 也能用 GPU。
- (3) 底层的 `tensor` 由 C 语言和 CUDA 技术实现，速度不会比 Caffe 差，某些运算甚至可能更快。
- (4) 使用 GPU 方便，把 `tensor` 数据送到 GPU 只需使用简单的“`tensor:cuda()`”命令。
- (5) Lua 入门快，堪比 Python。
- (6) 很重要的一点是，神经网络计算图理论上可以用神经网络里的模块实现任何 DAG 构造的网络，当然也包括 RNN、LSTM 之类的。

Torch 的缺点如下。

- (1) 对于不少人来说，要重新学习 Lua 语言。
- (2) 除了深度学习方面，其他好用的机器学习库较少。
- (3) 数据文件格式比较麻烦，一般原始数据没有 Torch 专用的 `t7` 格式文件，需要通过 `mat` 等格式进行转换。

1.4.4 DeepLearning4J

DeepLearning4J（简称 DL4J）是为 Java 和 Scala 编写的首个商业级开源分布式深度学习库，是由创业公司 Skymind 于 2014 年推出的，尤其值得注意的是，这家公司获得了腾讯千万级投资。DeepLearning4J 与 Hadoop 和 Spark 集成，为商业环境（而非作为研究工具）而设计。

DeepLearning4J 包括了分布式、多线程的深度学习框架，以及普通的单线程深度学习框架。定型过程以集群进行，也就是说，DeepLearning4J 可以快速处理大量数据。神经网络可通过（迭代化简）平行模型，与 Java、Scala 和 Clojure 兼容。DeepLearning4J 在开放堆栈中作为模块组件的功能，使之成为首个为微服务架构打造的深度学习框架。

DeepLearning4J 实现了受限玻尔兹曼机、卷积神经网络、递归神经网络、长短时记忆网络、自动编码器、深度信念网络（DBN）等常见深度网络模型。

DeepLearning4J 的优点如下。

- ❑ 功能多样的 n 维数组类，为 Java 和 Scala 设计。
- ❑ 与 GPU 集合。
- ❑ 可在 Hadoop、Spark 上实现扩展。
- ❑ Canova：机器学习库的通用向量化工具。
- ❑ ND4J：线性代数库，较 NumPy 快 1 倍。

DeepLearning4J 的缺点如下。

- ❑ 由于是创业公司新推出的产品，虽然不乏巨头采用，但是社区还是偏小。
- ❑ 例程不够丰富，虽然官方有一些例子，但是很多例子相对软件版本来说过于老旧，有一些模型还不能收敛。不像 Caffe 这种框架，提供 Model Zoo，可以选择前人已经预训练好的模型，不必重头开始训练网络，以节省工作量，可以直接应用训练好的模型，或者在训练好的模型基础上针对新问题进行迁移学习。

1.4.5 Caffe

Caffe 是一个清晰而高效的深度学习框架，其作者是毕业于 UC Berkeley 的贾扬清博士，其前身是贾扬清读博期间的研究项目。

Caffe 是纯粹的 C++/CUDA 架构，支持命令行、Python 和 MATLAB 接口；可以在 CPU 和 GPU 之间无缝切换（使用 `Caffe::set_mode(Caffe::GPU);`）；在 Caffe 中图层需要使用 C++ 定义，而网络则使用 Protobuf 定义。Caffe 是一个深度卷积神经网络的学习框架，使用 Caffe 可以比较方便地进行 CNN 模型的训练和测试，精于计算机视觉（CV）领域。

Caffe 作为快速开发和工程应用是非常合适的。Caffe 官方提供了大量实用的例程，通过这些例程可以实现各种常用功能。Caffe 只要求会写 prototxt 就行，它的训练过程、梯度下降算法等的实现都封装好了，学会 prototxt 的语法基本就能自己构造神经网络了。Caffe 作为 C++ 语言及配合了 CUDA 开发的框架，训练效率也有保证，这也是 Caffe 适

合工业应用的原因。代码易懂、好理解、高效、实用，上手简单，使用方便，比较成熟和完善，实现基础算法方便快捷，但是开发新算法不是特别灵活，所以比较适合工业快速应用实现。

Caffe 的优点如下。

- ❑ 在调参方面，改网络很方便，开源做得很好。
- ❑ 上手快，配置文件简单，文档齐全，模型与相应优化都是以文本形式而非代码形式给出的。
- ❑ 给出了模型的定义、最优化设置及预训练的权重，方便立即上手。这点其实是一个非常重要的优点，因为在实际的项目中，我们很少需要从头训练一个卷积神经网络，Caffe 允许我们在已经训练好的网络模型上进行微调和迁移学习，可以很好地解决实际项目中训练样本集小的限制条件。
- ❑ 速度快，Google Protocol Buffer 数据标准为 Caffe 提升了效率，能够运行最棒的模型与海量的数据。Caffe 与 cuDNN 结合使用，测试 AlexNet 模型，在 K40 上处理每张图片只需要 1.17ms。
- ❑ 模块化，允许对新数据格式、网络层和损失函数进行扩展，方便扩展到新的任务和设置上。
- ❑ 可以使用 Caffe 提供的各层类型来定义自己的模型。
- ❑ 开放性好，公开的代码和参考模型用于再现。
- ❑ 社区好，可以通过 BSD-2 参与开发与讨论。
- ❑ 学术论文采用此模型的较多。不少论文都与 Caffe 有关（如 R-CNN、DSN，还有人用 Caffe 实现了 LSTM）。

Caffe 的缺点是灵活性差，不同版本接口不兼容，可定制性较低，不能很方便地扩展到其他模型。

Caffe 可能是第一个主流的工业级深度学习工具，它开始于 2013 年年底，具有出色的卷积神经网络实现。在计算机视觉领域，Caffe 依然是最流行的工具包，它有很多扩展，但是由于一些遗留的架构问题，对递归神经网络和语言建模的支持很差。

1.4.6 MXNet

MXNet 结合了命令式和声明式编程的优点，既可以对系统做大量的优化，又可以方便调试。资源和计算的调度、内存分配资源管理、数据的表示、计算优化等都是很值得学习的，其原生支持分布式训练。

对于一个优秀的深度学习系统，或者更广泛地说，对于一个优秀的科学计算系统，最重要的是编程接口的设计。它们都将领域特定语言（Domain Specific Language）嵌入主语言中，例如 NumPy 将矩阵运算嵌入 Python 中。这类嵌入一般分为两种，一种嵌入得较浅，其中每个语句都按原来的意思执行，且通常采用命令式编程（Imperative Programming），

NumPy 和 Torch 就属于这种。另一种则用一种深的嵌入方式，提供一整套针对具体应用的迷你语言，通常使用声明式语言（Declarative Programing），即用户只需要声明要做什么，而具体执行则由系统完成。这类系统包括 Caffe、Theano 和 TensorFlow。

这两种方式各有利弊，总结如下。

1. 命令式编程

如果执行 $a=b+1$ ，需要 b 已经被赋值，立即执行加法，将结果保存在 a 中。

优点：语义容易理解、灵活，可以精确控制行为。通常可以无缝地和主语言交互，方便地利用主语言的各类算法、工具包、debug 和性能调试器。

缺点：实现统一的辅助函数和提供整体优化都很困难。

2. 声明式编程

如果执行 $a=b+1$ ，先返回对应的计算图，再对 b 进行赋值，然后执行加法运算。

优点：在真正开始计算的时候已经拿到了整个计算图，所以我们可以做一系列优化来提升性能。实现辅助函数也很容易，例如对任何计算图都提供 forward 函数和 backward 函数，对计算图进行可视化，将图保存到硬盘和从硬盘读取。

缺点：很多主语言的特性都用不上。某些特性在主语言中实现简单，但在这里却经常会遇到麻烦，例如 if-else 语句。debug 也不容易，例如监视一个复杂计算图中某个节点的中间结果就非常复杂。

目前现有的系统大部分都采用以上两种编程模式中的一种。与它们不同的是，MXNet 尝试将两种模式无缝结合起来。在命令式编程中 MXNet 提供张量运算，而在声明式编程中 MXNet 支持符号表达式。用户可以自由地组合使用它们来快速实现自己的想法。例如，我们可以用声明式编程来描述神经网络，并利用系统提供的自动求导来训练模型。而且，模型的迭代训练和更新模型法则中可能涉及大量的控制逻辑，因此我们可以用命令式编程来实现。同时我们用它来进行方便的调试并与主语言交互数据。

表 1.1 比较了 MXNet 和其他流行的深度学习框架。

表 1.1 深度学习流行框架特性比较

	Caffe	Torch	Theano	TensorFlow	MXNet
主语言	C++	Lua	Python	C++	C++
从语言	Python	Matlab	Python	Python	Python
硬件	CPU、GPU	CPU、GPU	CPU、GPU	CPU、GPU	CPU、GPU
分布式	否	否	否	是	是
命令式	是	是	否	否	是
声明式	否	否	是	是	是

MXNet 使用多值输出的符号表达式来声明计算图。符号是由操作子构建而来的，一个操作子可以是一个简单的矩阵运算“+”，也可以是一个复杂的神经网络里面的层，例如卷积层。一个操作子可以有多个输入变量和多个输出变量，还可以有内部状态变量。一个变

量既可以是自由的，之后对其赋值，也可以是某个操作子的输出。在执行一个符号表达式前，我们需要对所有的自由变量进行赋值。

【NDArray：命令式的张量计算】

MXNet 提供命令式的张量计算来桥接主语言和符号表达式。另外，NDArray 可以无缝地和符号表达式进行对接。

【KVStore：多设备间的数据交互】

MXNet 提供一个分布式的 key-value 存储来进行数据交换。它主要有两个函数：push 将 key-value 从一个设备 push 进存储；pull 将某个 key 上的值从存储中 pull 出来。此外，KVStore 还接受自定义的更新函数来控制收到的值如何写入存储中。而且，KVStore 提供数种包含最终一致性模型和顺序一致性模型在内的数据一致性模型。

【读入数据模块】

数据读取在整体系统性能上占重要地位。MXNet 提供工具能将任意大小的样本压缩打包成单个或数个文件，以加速顺序和随机读取。

【训练模块】

MXNet 实现了常用的优化算法来训练模型。用户只需要提供训练数据迭代器和神经网络的 Symbol 便可。此外，用户可以提供额外的 KVStore 来进行分布式训练。

【过去、现在和未来】

数个优秀的 C++机器学习系统的开发人员成立了 DMLC，初衷是更方便共享各自项目的代码，并给用户提供一致的体验。当时有两个深度学习的项目，一个是 CXXNet，其通过配置来定义和训练神经网络；另一个是 Minerva，提供类似 NumPy 的张量计算接口。前者在图片分类等使用卷积网络的应用上很方便，而后者更灵活。那时候这些开发人员想能不能做一个两种功能都具备的系统，于是就有了 MXNet，其名字取自 Minerva 中的“M”和 CXXNet 中的“XNet”。Symbol 的想法来自 CXXNet，而 NDArray 的想法来自 Minerva。MXNet 也常被叫作“Mix Net”。

MXNet 的目的是做一个有意思的系统，能够让大家用起来方便的系统，一个轻量且可以快速测试系统和算法想法的系统。未来主要关注下面四个方向。

- ❑ 支持更多的硬件，目前在积极考虑支持 AMD GPU、高通 GPU、Intel Phi、FPGA 和更多智能设备，相信 MXNet 的轻量和内存节省特性可以在这些方面大有作为。
- ❑ 更加完善的操作子。目前不论是 Symbol 还是 NDArray，它们支持的操作都很有有限，将来有望扩充它们。
- ❑ 更多编程语言。除了 C++，目前 MXNet 对 Python、R 和 Julia 的支持也比较完善。将来可能会支持更多的语言，例如 JavaScript。
- ❑ 更多的应用。之前花了很多精力在图片分类方向上，以后会考虑支持更多的应用。

1.4.7 CNTK

CNTK 是由微软公司推出的开源深度学习框架，其将公司的人工智能成果 CNTK 开源，并且放到 GitHub 上，自称是运算速度最快的工具集。

CNTK 是一个统一的深度学习工具包，它将神经网络描述成在有向图上的一系列计算步骤。在这个有向图中，叶子节点表示输入层或网络参数，其他的节点表示输入层上的矩阵操作。在 CNTK 上可以很容易地实现及结合当今流行的模型，例如前馈神经网络、卷积神经网络、循环神经网络。在实现随机梯度下降学习时能够自动计算梯度，而且还能通过多个 GPU 或服务器实现并行计算。CNTK 是微软在 Cortana 数字助理和 Skype 翻译应用中使用的语音识别系统框架。

CNTK 最大的优点是可以并行多个 GPU 或服务器。微软首席科学家黄学东说过：“谷歌公开的 TensorFlow 并没有这个功能。”

CNTK 的另外一个优点是支持 Microsoft Windows。但是这个开源工具是用 C++ 写的，微软计划尽快公开对应的 Python 和 C# 版本。

1.4.8 深度学习框架造型指导原则

在介绍了这么多开源深度学习框架之后，读者自然会有一个疑问，在这么多开源框架之中，到底应该选择哪个框架呢？和许多技术选型的问题一样，这个问题也没有标准答案，选择哪种方案都是各有利弊的，关键在于项目的具体需求。

在开源深度学习框架选择中，建议从以下四个方面来考虑。

1. 应用领域

这是首要因素，如果研究的是计算机视觉和图像处理领域，目前应用最成功的无疑是卷积神经网络，而对卷积神经网络支持最好的框架无疑要属 Caffe 框架。虽然其他框架同样支持卷积神经网络，但是 Caffe 框架提供了 2014 年和 2015 年 ImageNet 国际竞赛冠军所采用的网络架构的预训练模型，我们可以在其基础上进行微调或迁移学习，不仅提高了研发效率，同时也在一定程度上解决训练样本较少的问题。而且，目前很多图像处理领域的最新研究都是基于 Caffe 做出的，要想跟踪这些进展，采用 Caffe 框架也是比较方便的。

2. 项目性质

需要确定我们的项目是商业应用还是研究性质的。对于商业应用，一定要选择商用水平较高的框架，例如 TensorFlow、DeepLearning4J、Caffe 等。但是如果是一个前瞻性项目，需要反复试验各种不同的模型，选择 Torch 等比较灵活高效的框架是不错的选择。如果项目是研究性质的，需要跟踪行业的最新进展，与其他顶级研究人员交流，那么 Theano 框架就是最佳选择。

3. 效率、速度、性能

这里的效率、速度、性能主要指训练阶段，因为神经网络在运行时还是相当快的，但是在训练阶段，迭代运算往往需要很大的计算量。以利用 ImageNet 数据集训练卷积神经网络为例，如果采用 AlexNet 或 GoogleLeNet 架构，即使在高端机器上也需要运行两三周，在普通机器上时间就会更长。选择性能高的框架，意味着在单位时间内，可以尝试更多的模型或超参数调优，更有利于出成果。在性能方面，Caffe、MXNet、CNTK、Torch 等均有非常不错的表现，在选型时可以加以考虑。

4. 学习曲线

我们还需要考虑入门难易程度、文档的完善程度、社区的活跃度等问题。从入门方面来看，Torch 由于封装得较好，入门相对容易。人们对 TensorFlow 的关注程度较高，有丰富的教程，学习曲线也较平缓。另外，还可以考虑采用 Keras 等框架，其在 Theano 等框架的基础上进行了封装，大大降低了学习难度。

1.5 深度学习入门路径

深度学习概念火爆之后，很多人都想进入深度学习领域，但是深度学习领域由于理论门槛较高，不像互联网、移动互联网技术那样可以很快地进入。大家都急切地想知道，深度学习到底怎样才能快速入门。

由于深度学习技术需要高深的数学知识作为支撑，因此有人建议先打好数学基础和机器学习的基础，再来学习深度学习技术。但是从高等数学、线性代数、概率论与数理统计开始，然后是机器学习、神经网络，最后到深度学习，如果只是利用业余时间自学的话，这个过程少说也得三五年，而三五年之后，深度学习是否还是现在这个样子就很难说了。所以还是建议读者先掌握深度学习的大体框架，对每种深度学习技术有一个大概的了解，然后根据自己感兴趣的方向选择一种适合的技术，再根据技术研究的需要，随时补充自己欠缺的知识，这样学习效率会更高一些。

下面是对学习深度学习入门路径的建议，读者可以参考这个路径尽快进入深度学习领域，并在各自的领域内应用深度学习技术。

1.5.1 运行 MNIST

学习任何一项新技术的第一步都是将这个技术运行起来。在学习深度学习的过程中，这一步并不难，因为正如 1.4 节所介绍的，市面上有非常多的开源框架，随便选择一种框架，再找一个教程，就可以把框架提供的例子跑起来。这样做的好处有两个，一方面通过自己

亲手运行一个深度学习程序，给自己增加信心；另一方面，通过这个例子，也会对深度学习过程有一个感性的认识。

但是在这个过程中，选择合适的例子很重要。选择太简单的例子，会看不出任何实际应用的背景，提不起我们的兴趣。更重要的是，如果选择的是类似求解异或的这类问题，虽然可以演示求解非线性问题，但是因为没有训练样本集、验证样本集、测试样本集的区别，同时没有欠拟合或过拟合的风险，实际上还是没有了解深度学习的全貌。

同样，我们也不要选择过于复杂的例子，过于复杂的例子在细节上非常烦琐，很容易使我们陷入不必要的细节中，从而忽略了对主要问题的把握。

综上所述，我们认为 MNIST 手写数字识别是一个非常不错的例子。首先，这个例子相对简单，但却是具有实际应用价值的例子，可以提起我们的兴趣；其次，这个例子虽然简单，但是里面包括了训练样本集、验证样本集、测试样本集，同时也存在着欠拟合和过拟合问题，基本包括了深度学习的主要方面；最后，这个例子也是日后我们自己做学习算法的一个衡量标准，我们的算法表现到底如何，用 MNIST 手写数字识别数据测试一下，基本就可以有一个大致的了解了。

编译运行完这些开源框架提供的例子，当产生正确的输出结果之后，我们通常会非常兴奋，觉得自己终于掌握深度学习技术了。但是初始的兴奋劲过去之后，不禁陷入了迷茫，除运行这些例子及看到运行结果之外，我们还能干什么呢？从哪里入手呢？

首先，各种开源框架的例子虽然有所不同，但是基本都有一个共同特点，那就是其只讲了如何训练网络，但是网络是要实际运行，而不是只训练一下就行的，怎么使例子变为运行状态呢？即网络只进行模式识别，而不像训练阶段那样利用学习算法修改参数。如果能达到这一步，就说明我们基本搞明白框架的运行机制了。

最后，MNIST 数据集是手写的数字识别数据集，我们能不能提供自己的手写数字图片让训练好的网络来识别呢？这就需要我们深入了解 MNIST 数据集的格式，或者开源框架对 MNIST 数据集预处理后的格式，搞明白这些之后，就可以将图片包装成 MNIST 格式，让训练好的网络来识别了。

当我们能够顺利完成上述工作之后，就可以认为真的入门了，为将这个开源框架应用到实际项目中打下了基础。

1.5.2 深度学习框架的选择

在 1.4 节中，我们介绍了几个主流的深度学习开源框架，分别介绍了其使用场景、优缺点和选择过程中需要考虑的要素。但是由于所有这些讨论都没有基于实际的例子，所以可能比较抽象，读者感觉难以理解。在这里，我们假设要处理的任务是图像处理，包括图像识别、物体检测、图像分割、图像标注等内容，下面来看看应该选择什么样的开源框架。

为了方便起见，下面把几个常用的开源框架 Caffe、Torch、Theano、TensorFlow 以表格的形式列了出来，根据我们的项目特点分别进行讨论，如表 1.2 所示。

表 1.2 主流的深度学习开源框架优缺点汇总

框架名称	优 点	缺 点
Caffe	<ul style="list-style-type: none">• 擅长前向网络• 擅长调优现有网络，大量预训练网络• 模型训练不用编写代码• 完善的Python接口• 代码可读性好	<ul style="list-style-type: none">• 对于新的层需要写C++/ CUDA代码• 不适合递归神经网络• 模型文件对于大型网络（如GoogleLeNet）等来说过于庞大难读
Torch	<ul style="list-style-type: none">• 搭积木式构建神经网络• 定义新网络层非常简单，并且可以自动在GPU上运行• 预训练模型丰富	<ul style="list-style-type: none">• 采用小众的Lua语言• 训练网络需要写Lua代码• 不适合开发递归神经网络
Theano	<ul style="list-style-type: none">• Python+NumPy组合• 非常好的计算图抽象实现• 适合开发递归神经网络（RNN）• 第三方扩展简化开发过程，如Keras和Lasagne• 深度学习三巨头之一Bingio大力支持	<ul style="list-style-type: none">• 直接使用比较复杂• 出错信息难以调试，因为函数采用预编译，不能单步跟踪• 大型模型的编译时间长• 预训练模型支持不好
TensorFlow	<ul style="list-style-type: none">• Python+NumPy组合• 计算图抽象实现• 利用TensorBoard可以实现可视化• 数据和模型可以在多GPU上并行，在所有框架中最强• 分布式模型（但是未开源）• 背后有谷歌支持	<ul style="list-style-type: none">• 比其他框架慢• 预训练模型少• 体积比较大

下面是这四种开源框架的特性对比，如表 1.3 所示。

表 1.3 主流的深度学习开源框架特点总结

	Caffe	Torch	Theano	TensorFlow
语言	C++ 、 Python	Lua	Python	Python
预训练模型	丰富	丰富	有（Lasagne）	少Inception
多GPU数据并行	有	有	有	有
多GPU模型并行	无	有（fbcunn.ModelParallel）	试验性	强项
代码可读性	强（C++）	强（Lua）	弱（Python）	弱（Python）
RNN支持	弱	中等	强	强

项目 1：抽取图像特征用于模式识别

假设目前的项目需要识别出图像的特征，并用这些特征来做模式识别工作，例如拿肺部胸片判断患者是否患有肺癌，则可以通过对图像特征的学习来对胸片进行诊断。首先要做的就是抽取图像的特征，由于所有的胸片数据有几万张图片，而胸片图片尺寸又非常大，训练全新的网络会面临训练样本严重不足的情况。

针对这种情况，建议采用 Caffe。因为 Caffe 有着丰富的预训练模型，例如 AlexNet、

VGG、GoogleLeNet、ResNet 等，都可以拿来即用，这些预训练好的网络，已经经过 1200 万张图片的训练过程，对于特征抽取来说是非常高效的。

另外，虽然可以直接使用 Caffe 预训练模型中的特征，但是即使经过卷积神经网络提取到的特征，也可能存在大量冗余。因此在数据输入到训练网络之前，先利用主成分分析 (PCA)、自动编码器、受限玻尔兹曼机等对这些特征进行预处理，去掉冗余信息，则可以取得更好的应用效果。

实际工作流程如图 1.1 所示。



图 1.1 图像特征抽取工作流程

项目 2：网络微调应用

假设项目是要用图像识别系统进行名贵狗品种的鉴别。通过狗的照片给出这是什么品种的狗、是否是纯种狗，以及是否是不同品种的狗杂交得到的新品种。这时我们有几千张各种名贵狗的图片可以作为训练样本数据。

由于要应用的领域与 ImageNet 非常相似，ImageNet 本身就包含大量狗的图片，对狗的分类也做得比较好，因此用一个训练好的 ImageNet 优秀模型将是非常好的决定。

建议采用 Caffe 框架，先从 Model Zoo 中选择 GoogleLeNet、VGG 或 AlexNet 预训练模型，将其输出层替换为我们的分类器，例如 Softmax 回归层，再直接连接到原来的全连接层，然后拿我们的几千张图片组成的训练样本集对网络进行调优训练，最后得到可以用于实际应用的生产网络。

项目 3：图像标注应用

假设项目是要对 X 光图片进行标注，就是在图片上写上“双肺见絮状团块，疑似间质性病变”等这些我们常见的影像科诊断信息。这个问题实际上由两个阶段组成，第一个阶

段是图像特征抽取，第二个阶段是利用图像特征进行标注工作。

这个项目的工作流程如图 1.2 所示。

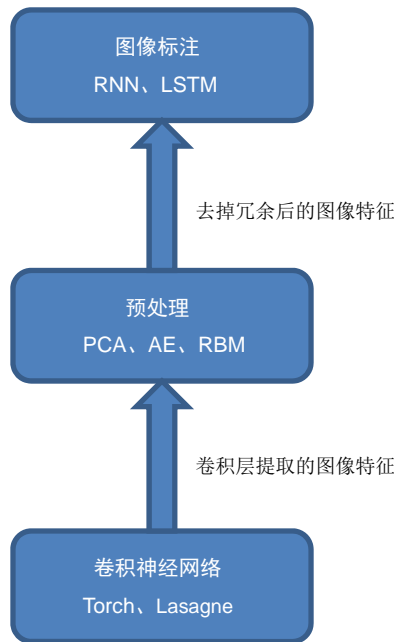


图 1.2 图像标注模型选择

在这个项目中，由于我们的训练样本集也很小，所以需要预训练的卷积神经网络模型。但是这与项目 1 还是有所不同的：不仅样本少，而且样本与原始的 ImageNet 图片差异非常大，这就要求我们提取图像特征的层应该是更底层一些的卷积层的输出。同时，由于后续工作是图片标注，需要使用递归神经网络或长短时记忆网络。综合考虑上面两方面的因素，建议项目采用 Torch 或 Lasagne（基于 Theano），因为这两个框架都有预训练好的网络，同时对递归神经网络或长短时记忆网络支持较好。

项目 4：图像分割

假设在这个项目中需要将图片中的猫、狗等从图片中抠出来，因此需要基于每像素来判断其属于猫或狗，还是属于背景。我们同样只有几千张图片的训练数据。

所以在这个训练中，还是需要预训练好的卷积神经网络模型，因此我们的选择就集中在 Caffe、Torch、Lasagne（基于 Theano 的扩展）。因为需要做图片分割，所以需要有不同的代价函数。如果这种代价函数在 Caffe 中有相应的实现，那么我们就选择 Caffe 框架；如果没有，则可以选择使用 Torch 的 Lua 进行开发。因为 Caffe 的 C++ 开发效率和安全性方面都可能存在比较大的隐患。

项目 5：目标物体检测

这个项目的目的是识别并标记图片中的物体，如人物、猫和狗等，并且训练样本也较少。这时同样需要预训练网络模型，同时由于要进行物体识别，可能还需要利用命令式编

程写一些较为复杂的逻辑。如果仅从预训练模型来看，我们可以选择 Caffe、Torch 和 Lasagne（基于 Theano 的扩展），但是因为要进行命令式编程，Theano 的计算图模型就显得特别不方便，所以最好不要选择 Lasagne。因此在这样的应用场景下，建议选择 Caffe，实际上目前有很多基于 Caffe 的物体检测 Fork，采用 R-CNN、Fast-RCNN、Faster-RCNN、CRF-CNN 等模型，可以直接拿来使用。不过可能这些模型使用的是较老版本的 Caffe，升级 Caffe 的因素也需要考虑。如果没有现成的模型可用，建议用 Caffe 的 Python 接口来写特殊的业务逻辑，因为这样开发效率较高，而且代码实现更安全。

综上所述，在图像识别领域，如果需要预训练模型，并且只用到了卷积神经网络，通常建议采用 Caffe 框架。如果同时还需要其他模型，如在图像标注应用中需要用到递归神经网络或长短时记忆网络模型，这时可以选择 Lasagne（基于 Theano 的扩展）。

1.5.3 小型试验网络

深度学习系统运算量非常大，下面以采用 GoogleLeNet 卷积神经网络架构对 ImageNet 训练数据集进行训练为例。在强大的云端服务器配置有顶级 GPU 的情况下，也需要连续运行两三周的时间。虽然我们的问题也许没有 ImageNet 图像识别那么复杂，但是大型网络的运行时间与这个例子应该是在同一数量级的，因此这是一个非常耗时的工作。一旦网络架构和超参数等选择失误，甚至是初始化权值不合理，我们就要浪费大量的人力、物力，因此在正式训练生产网络之前，首先要保证模型和超参数的合理性，这是一个非常重要的问题。

对于这个问题，应该首先应用已选择的开源框架建立一个小型试验网络，然后找一个与自己的研究问题类似的小型数据集，对小型试验网络进行训练，调整超参数，观察对网络性能的影响，如学习优化算法、代价函数、学习率、学习率衰减、神经网络层次、每层神经元数量、调整项 L1/L2、调整项系数等，通过试验找到这些参数的合理值。

同时，需要测试训练样本集与验证样本集的比例，找到最佳早期停止的策略、Dropout 的比例等。

当在小型数据集上完成上述这些工作之后，我们才可以说有了一个相对合理的深度学习解决方案，可以将这个方案应用到实际工作中。

1.5.4 训练生产网络

当小型试验网络在小数据集上得到验证之后，就可以拿真实的生产数据来训练生产网络了。在训练过程中需要注意，按照研究问题的性质，将数据集分为训练样本集、验证样本集、测试样本集，要尽量保证这 3 个样本集属于同一个概率分布，这样才不会出现训练样本集上表现很好，在验证和测试样本集上表现很差的情况。

另外，可以根据需要对原始输入数据进行预处理，将输入数据进行规整化，必要时可

以对网络隐藏层信号进行规整化，假设训练样本集用 S 表示，共有 $|S|$ 个样本，则先求出样本的期望：

$$E = \frac{1}{|S|} \sum_{x \in S} x_i$$

接着求出信号的方差：

$$\sigma = \sqrt{\frac{1}{|S|} \sum_{x \in S} (x_i - E)^2}$$

由此得到新的输入信号为：

$$\hat{x}_i = \frac{x_i - E}{\sigma}$$

从理论上可以证明，这样做并没有减少信号中的信息量，只是将输入信号移动到关于零点对称分布，并减小了信号的绝对值。我们知道，在神经网络中大量使用了 **Sigmoid** 或双曲正切函数，而这些函数只有在零点左右较小的区域内才会有比较好的表现，在其他区域曲线将出现饱和，学习速率会变得相当慢。上面的规整化，正好可以有效地解决这一问题。

在实际应用中，遇到的非常多的问题就是过拟合问题，这时虽然在训练样本集上有非常好的表现，但是在测试样本集或在实际应用中会出现比较大的误差。这通常是由于所选取的模型比当前要解决的问题所需要的表现能力高很多，模型为了完美地适应训练样本集才出现了过拟合，通常通过早期停止和 **Dropout** 来解决这一问题。但是，无论是从理论上还是实际上均可证明，适当复杂的网络模型解决问题的能力更强，因此我们应该倾向于使用稍微复杂的模型，并通过调整项来解决过拟合问题。

当然，神经网络的训练过程，包括超参数的选择，甚至算法的选择，目前仍是一门经验和艺术的工作，还没有公认的科学的方法和实践。因此在这个阶段，除进行理论上的分析之外，更多的是在实践中根据实验数据的反馈不断地调整和完善网络，直到达到希望的效果为止。

1.5.5 搭建生产环境

训练好的神经网络，就可以部署到实际生产环境，应用于实际项目了。但是我们的神经网络系统是一个计算密集型业务，所以最好部署到具有 **GPU** 的计算集群上，而且我们的网络可能涉及持续调优，需要使用大量数据，所以部署的位置理应放在离数据最近的位置上。

1.5.6 持续改进

已经投入实际使用的神经网络系统还需要不断完善，使其性能不断提高，能够持续满足我们的需求。但是这通常只是一个美好的愿望，因为对神经网络进行调优比训练神经网络要难得多，在这方面更加缺乏有效的理论和实践支持。通常可以关注以下几个方面。

首先，对于由于训练样本集不够大，影响到网络性能的情况，因为在实际应用中会持续产生新的样本，可以考虑在收集足够多的新样本之后，重新对网络进行训练，或者在目前的网络基础上再次对网络进行训练。可以采用以下方式进行：在网络运行一段时间之后，先收集足够的新样本，将原来的验证样本集合并到训练样本集，然后将收集到的新样本作为验证样本集，对网络进行重新训练，最后在测试样本集上查看训练效果，如果有改进则部署到生产环境，按照这种方式可以持续改进生产模型。

其次，深度学习领域目前还处在高速发展阶段，新方法、新实践层出不穷，经常有文章报道采用某种改进的算法将原来的性能提高了多少。因此需要时刻关注自身领域的最新进展，不断尝试新的方法，并在训练样本集、验证样本集、测试样本集上进行试验，发现有显著改进时，就将其更新到生产环境。

总之，将神经网络部署到生产环境中后，我们的工作并没有因此而结束，还需要持续改进模型，为获取更好的性能而努力。另外，我们有训练样本集、验证样本集、测试样本集可以进行回归测试，可以非常方便地比较各种改进的优劣，减少因为改动使网络性能恶化的风险。

第二部分 深度学习算法基础

- ☐ 搭建深度学习开发环境
- ☐ 逻辑回归
- ☐ 感知器模型和 MLP
- ☐ 卷积神经网络
- ☐ 递归神经网络
- ☐ 长短时记忆网络

第 2 章

搭建深度学习开发环境

2.1 安装 Python 开发环境

在本章中，我们将以 Ubuntu 系统为例，讲述 Python 系统配置过程。

2.1.1 安装最新版本 Python

目前 Ubuntu 系统自带的 Python 系统还是 2.7.x 版本的，而我们在本书中将采用 Python 3.5 版本，所以需要安装最新的 Python 系统。下面将讲述从 Python 源码编译安装最新版 Python 系统的步骤。

首先，从 Python 官方网站下载最新版本的 Python 源码：

```
wget https://www.python.org/ftp/python/3.5.2/Python-3.5.2.tgz
```

新建一个目录，将 Python-3.5.2.tgz 复制到该目录，并解压缩该文件：

```
tar -xvzf Python-3.5.2.tgz
```

系统会将源码解压缩到目录 Python-3.5.2，进入该目录，配置编译选项：

```
./configure --prefix=/home/osboxes/dev/python352 --enable-shared CFLAGS=-fPIC
```

在这里指定了 Python 系统的安装目录，并且允许使用动态链接库，系统将自动根据当前系统环境配置，生成相应的 Makefile 文件。编译 Python 源码命令为：

```
make
```

最后，安装新编译出来的 Python 系统：

```
make install
```

由于系统的环境配置不同，Python 系统可能没有被成功地安装到系统目录中去，所以可能需要执行以下命令：

```
export LD_LIBRARY_PATH=/home/osboxes/dev/python352/lib:$LD_LIBRARY_PATH
```

2.1.2 Python 虚拟环境配置

在大型软件项目开发过程中，经常会用到各种库，而不同的项目对同一个库的版本要求可能是不同的，因此经常出现库版本冲突的问题。

Python 对于这种情况提出了一种高效的解决方案，即每个项目都可以创建一个虚拟环境，在这个虚拟环境中，安装项目所需要的特定版本的依赖库，并且将项目所需要的所有库统一通过配置文件进行管理。这样当需要将系统从开发环境部署到生产环境时，只需要在生产环境中建立相应的虚拟环境，用开发环境的配置文件进行初始化，就完成了整个部署过程。同时，即使有多个系统使用相互冲突的库，由于各个系统都在自己的虚拟环境中运行并调用自己虚拟环境中的库，所以也不会产生冲突。

在 Python 2.x 时代，通常会安装 `virtualenv` 这一工具来建立虚拟环境，但是在 Python 3.x 版本中，系统自带了一个类似的工具 `venv`，其使用方法和功能与 `virtualenv` 完全相同，用户就不用再安装额外的软件了。

当然，为了处理各个安装部署环境不同引起的问题，除了 Python 所提供的虚拟环境解决方案，还有其他的解决方案，如 `Vagrant` 基于虚拟机的解决方案。通过 `VirtualBox` 建立虚拟机，使开发人员都在虚拟机上进行开发，统一大家的环境；还有 `Docker`，通过容器技术简化安装部署过程。但是在本章中，我们将主要讨论对 Python 虚拟环境的使用，其他技术暂不进行讨论，有兴趣的读者可以到网上参考相关资料。

首先确保已经将编译好的 Python 放到系统路径上：

```
export PATH=/python 安装目录:$PATH
```

这种方式只在当前终端中起作用，重新打开终端时就会失效。如果想让对路径的修改永久生效，可以修改当前用户的环境变量，运行：

```
sudo vim .profile
```

在文件中加入以下内容：

```
export PATH=/python 安装目录:$PATH
```

然后使修改的内容生效：

```
source ./profile
```

进行了上面的准备工作，就可以开始创建项目的虚拟环境了。进入将要创建项目的工作目录，运行以下命令：

```
pyvenv proj
```

其中，`proj` 是你要创建的项目的名称，也是路径名称。Python 为我们在当前目录下创

建了一个名为 `proj` 的目录，进入该目录你会发现 Python 在这里创建了一个单独的 Python 运行环境，目录结构如图 2.1 所示。

```
total 24
drwxrwxr-x 5 osboxes osboxes 4096 Nov 6 01:36 .
drwxrwxr-x 3 osboxes osboxes 4096 Nov 6 01:36 ..
drwxrwxr-x 2 osboxes osboxes 4096 Nov 6 01:36 bin
drwxrwxr-x 2 osboxes osboxes 4096 Nov 6 01:36 include
drwxrwxr-x 3 osboxes osboxes 4096 Nov 6 01:36 lib
lrwxrwxrwx 1 osboxes osboxes   3 Nov 6 01:36 lib64 -> lib
-rw-rw-r-- 1 osboxes osboxes  91 Nov 6 01:36 pyvenv.cfg
```

图 2.1 `proj` 目录结构

到目前为止，我们就为项目创建了一个虚拟环境，并且将在这个环境中进行后续的程序开发。

2.1.3 安装科学计算库

要使用 Python 进行深度学习开发，需要安装一些依赖库，下面在刚建好的虚拟环境中安装这些依赖库：

```
sudo apt-get install gfortran
sudo apt-get install libopenblas-dev
sudo apt-get install liblapack-dev
sudo apt-get install libatlas-base-dev
```

确定依赖库安装在刚刚创建好的虚拟环境目录下后，先激活虚拟环境：

```
#source ./bin/activate
```

然后安装 Python 依赖库：

```
pip install nose
pip install numpy
pip install scipy
```

至此，我们就将 Python 科学计算应用中所需要的依赖库安装完成了，下面就可以安装我们需要的深度学习框架 Theano 了。

2.1.4 安装最新版本 Theano

如果前面 Python 及依赖库的安装没有出现问题，安装 Theano 就变成一件很简单的事情，只需要运行：`#pip install theano` 就可以将 Theano 框架安装完成。

2.1.5 图形绘制

如果要使用 Python 进行图形绘制，在使用源码安装最新版本 Python 之前，必须先安装

Python 绘图的依赖库。如果已经编译安装了 Python，那么就需要在安装完依赖库之后重新编译安装 Python 系统。

(1) 检查环境是否安装了 `_tkinter`，运行命令如下：

```
>>> import _tkinter
```

如果没有报错，就可以进行第 2 步操作了；如果报错，需要配置一下 `_tkinter` 库。安装依赖库的命令如下：

```
sudo apt-get install libx11-dev
sudo apt-get install blt-dev
sudo apt-get install python-tk
```

运行上述命令时，会安装各自的依赖库，主要包括 `tck` 和 `tk`，需要确认安装。

安装完成之后，需要重新编译安装 Python，在 Python 的源码目录下：

```
make clean
make distclean
./configure --prefix=/home/osboxes/dev/python352 --enable-shared CFLAGS=-fPIC
make
make install
```

由于我们没有将 Python 安装到默认目录，所以需要配置库路径，编辑 `vim ~/.profile` 文件，在最后面添加：

```
export LD_LIBRARY_PATH=/home/osboxes/dev/python352/lib:$LD_LIBRARY_PATH
```

然后使其生效：

```
source ~/.profile
```

重新建立虚拟环境：

```
/home/osboxes/dev/python352/bin/pyvenv wky
```

上面的路径是我们安装的新版 Python 的路径，会在当前目录下建立 `wky` 目录，该目录下即我们的虚拟环境。

进入该目录，激活虚拟环境：

```
source ./bin/activate
```

这时进入 Python 交互环境，再输入 `import _tkinter` 命令时就应该不报错了，这证明我们已经成功安装配置了 `tkinter`。

(2) 检查 `tkinter`。

在 Python 交互环境下输入 `import tkinter`，应该保证系统不报错。

(3) 测试 `tkinter`。

紧接上一步，输入 `tkinter._test()`，如果系统弹出一个窗口，上面会有两个按钮，一个是“click me”，另一个是“quit”。单击“click me”按钮，其会变为“[click me]”按钮，这就证明 Python 绘图环境的配置是正确的，可以单击“quit”按钮退出了。

我们在 Python 图形编程中，经常会使用一个类似 Matlib 的绘图库，可以使用类 Matlib 方法来进行图形绘制，这使机器学习中的图形绘制也非常方便，所以需要安装这个图形绘制库：

```
pip install matplotlib
```

在机器学习中，Logit 函数是一个经常会被用到的函数，一般神经网络的输出层均采用这个函数作为激活函数，这个函数的定义域为 $(-\infty, +\infty)$ ，值域为 $(0,1)$ ，可以视其为概率，函数表达式为：

$$y = \frac{1}{1 + e^{-x}}$$

通过 matplotlib 绘制函数图像就变为一个相对简单的过程了，代码如下：

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.linspace(-5.0, 5.0, 1000)
5 y = (1 / (1 + np.exp(-x)));
6 plt.figure(figsize=(8, 4))
7 plt.plot(x, y, label="$exp(x)$", color="blue", linewidth=2)
8 plt.xlabel("x")
9 plt.ylabel("y")
10 plt.title("Logistic function")
11 plt.ylim(0, 1.1)
12 plt.xlim(-5.0, 5.0)
13 plt.legend()
14 plt.show()
15
```

- 第 1、2 行：引入绘图所需的库。
 - 第 4 行：定义 x 取值范围为-5~5，被分为 1000 段。
 - 第 5 行：定义 Logit 函数。
 - 第 6 行：定义一个图像，分辨率为 800×400。
 - 第 7 行：定义绘制对象，标题为 $\exp(x)$ ，定义线条为“蓝色”，宽度为“2”。
 - 第 8、9 行：指定 X 轴、Y 轴的标题。
 - 第 10 行：定义图像的标题。
 - 第 11、12 行：定义 X 轴和 Y 轴的取值范围。
 - 第 13 行：定义图例。在有多个曲线时，标注每个曲线代表的含义将非常有用。
 - 第 14 行：显示图像。
- 运行上面的程序，会得到如图 2.2 所示的图形。

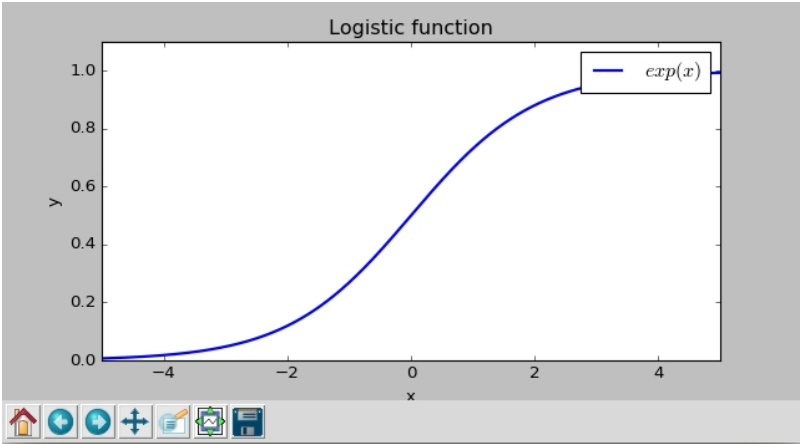


图 2.2 Logit 函数图形

由上面的例子可以看出，利用 Python 绘制函数图形还是非常方便的。这对于深度学习中的数据可视化方面来说，是一个非常好的工具，如果读者对这方面感兴趣，可以在网上找到很多资料，这里就不详细介绍了。

2.2 NumPy 简易教程

由于 Theano 框架中会大量应用 NumPy 库，如当进行数组、向量、矩阵、张量计算时，都会调用 NumPy 的功能来实现。因此在本节中，先简单向读者介绍 NumPy 的基本功能和用法，为后续的 Theano 框架学习打好基础。

2.2.1 Python 基础

NumPy 库的基础是 Python，所以先来看一下 NumPy 中经常用到的关于 Python 的知识点。我们知道 Python 是动态、强类型语言，也是支持面向对象和函数式编程的高级语言，语言非常简洁灵活，涉及的内容很广。本书不详细介绍 Python 的语言特性，因此主要讲解一下在深度学习领域使用最多的数据类型和容器类。

1. 基本数据类型

1) 数字类型 (Numbr)

在 Python 中的整数、浮点数统称为数字类型，使用方法如下：

```
1 x = 3 # 声明整数变量
2 print type(x) # 会打印 "<type 'int'>"
3 print x + 1 # 加法运算
4 print x - 1 # 减法运算
5 print x * 2 # 乘法运算
6 print x / 2 # 数学除法运算，此例中结果为1.5
7 print x // 2 # 整数除法运算，此例中结果为1
8 print x % y # 取余数
9 print x ** 2 # 平方
10 print y = 2.5
11 print type(y) # 打印 "<type 'float'>"
12 print abs(-x) # 取绝对值函数
13 print int(y) # 转换为整数
14 print long(x) # 转换为长整数
15 print float(x) # 转换为浮点数
```

2) 布尔类型 (Boolean)

Python 提供了布尔类型，并提供常量 True 和 False。与其他语言不同的是，Python 使用 and 或 or 而不是 && 或 || 来表示逻辑运算，代码如下：

```
1 v1 = True # 布尔常量：真
2 v2 = False # 布尔常量：假
3 print(type(v1)) # 打印："<type 'bool'>"
4 print((v1 and v2)) # v1 && v2, 结果为假
5 print((v1 or v2)) # v1 || v2, 结果为真
6 print((not v1)) # !v1, 结果为假
7 print((v1 != v2)) # 逻辑异或操作
```


3) 字符串类型（string）

Python 字符串操作功能非常强大，是 Python 传统的应用领域之一。在这里不准备介绍 Python 字符串中功能最强大的部分正则表达式，只介绍字符串的基本操作和常用函数。

Python 字符串的常用操作如下：

```
1 v1 = 'hello'
2 v2 = 'world'
3 print(v1)
4 print('%s' % v1)
5 print('string "%s" len=%d' % (v1, len(v1)))
6 v3 = v1 + ' ' + v2 + '!'
7 print(v3)
```

Python 提供了丰富的字符串常用函数，代码如下（chp02/c003_1.py）：

```
1 v1 = 'hello world, hello every one'
2 print('len=%d' % len(v1))
3 print('first=%d; second=%d' % (v1.find('hello'), v1.find('hello', 6)))
4 name = 'PI'
5 value = 3.1415926
6 print('{0}={1}'.format(name, value))
7 v2 = 'abcdefi8383'
8 print('isalnum=%s' % v2.isalnum())
9 v3 = 'abcd'
10 print('isalpha=%s' % v3.isalpha())
11 v4 = '123'
12 print('isnumeric=%s' % v4.isnumeric())
13 v5 = 'hello lady and gentel men'
14 print('new:%s' % v5.replace('lady', 'ladies'))
15 v6 = '1, 2, 3, 4, 5'
16 v7 = v6.split(',')
17 print('split:%s; type=%s' % (v7, type(v7)))
```

程序运行结果：

```
len=28
first=0; second=13
PI=3.1415926
isalnum=True
isalpha=True
isnumeric=True
new:hello ladies and gentel men
split:['1', ' 2', ' 3', ' 4', ' 5']; type=<class 'list'>
```

第 2 行：函数 len 返回字符串长度。

第 3 行：字符串函数 str.find(sub[, start[, end]]），其中 sub 为要查找的字符串，start、end 为可选参数，规定查找的范围。

第 6 行：演示字符串 format 函数用法，格式化字符串中可用 {i} 来代表变量列表中的变量，i 为变量列表中的索引值。

第 8 行：isalnum 判断字符串中是否仅有字符和数字。

第 10 行：判断字符串中是否只有字符。

第 12 行：判断字符串中是否只有数字。

第 14 行：str.replace(old, new[, count]) 函数将旧的子串用新的子串替代，默认为全文替代，可选参数 count 表示只替换前 count 个字符。

第 17 行：str.split(sep=None, maxsplit=-1) 表示用 sep 来分割，分割结果以 list 形式返回。

Python 中提供的容器类主要包括列表、字典、集合、元组，下面分别进行介绍。

2. 容器类之列表

Python 类的列表与其他语言中的数组类似，但是其大小可变，而且可以包含不同类型的元素。列表的相关函数如下（chp02/c004.py）：

```
1 xs = [5, 1, 8, 9, 6]
2 print('xs:%s' % xs)
3 print('xs[2]=%d' % xs[2])
4 print('xs[-1]=%d, xs[-2]=%d' % (xs[-1], xs[-2]))
5 xs[4] = 'six'
6 xs.append('seven')
7 print('xs:%s' % xs)
8 top = xs.pop()
9 print('top=%s, xs:%s' % (top, xs))
```

程序运行结果：

```
xs:[5, 1, 8, 9, 6]
xs[2]=8
xs[-1]=6, xs[-2]=9
xs:[5, 1, 8, 9, 'six', 'seven']
top=seven, xs:[5, 1, 8, 9, 'six']
```

第 1 行：初始化新列表。

第 2 行：打印列表内容。

第 3 行：打印列表中的某个元素。

第 4 行：列表序号也可以为负值，-1 代表最后一个元素，-2 代表倒数第二个元素。

第 5 行：列表元素可以是不同类型，在本例中，把最后一个元素变为字符型。

第 6 行：向列表末尾添加字符型元素。

第 7 行：在打印当前列表内容时，可以发现列表前 4 个元素为整数型，而最后两个元素为字符型。

第 8 行：取出列表最后一个元素，并从列表中删除该元素。

第 9 行：打印取出的最后一个元素和当前数组内容。

Python 提供了强大的功能，让我们可以非常方便地取出子列表，即切片，代码如下（chp02/c005.py）：

```
1 xs = [x for x in range(1, 5+1)]
2 print('xs:%s' % xs)
3 print('xs[2:4]=%s' % xs[2:4])
4 print('xs[2:]=%s' % xs[2:])
5 print('xs[:2]=%s' % xs[:2])
6 print('xs[:]=%s' % xs[:])
7 print('xs[:-1]=%s' % xs[:-1])
8 xs[2:4] = [101, 102]
9 print('xs:%s' % xs)
```

程序运行结果：

```
xs:[1, 2, 3, 4, 5]
xs[2:4]=[3, 4]
xs[2:]=[3, 4, 5]
xs[:2]=[1, 2]
xs[:]=[1, 2, 3, 4, 5]
xs[:-1]=[1, 2, 3, 4]
xs:[1, 2, 101, 102, 5]
```

第 1 行：利用 range 初始化列表。

第 2 行：打印列表内容。

第 3 行：从下标 2 开始取值，一直取到下标 4，但是不包括下标 4。

第 4 行：从下标 2 开始取值，一直取到结束。

第 5 行：从开头开始取值，一直取到下标 2，但是不包括下标 2。

第 6 行：不指定数字，表示取整个列表。

第 7 行：从开头开始取值，一直取到最后一个，但是不包括最后一个。

第 8 行：将从下标 2 开始（包括）一直到下标 4（不包括）的内容，替换为新的列表内容。

循环列表内容（chp02/c006.py）：

```
1 xs = ['one', 'two', 'three']
2 for item in xs:
3     print('%s' % item)
4
5 for idx, item in enumerate(xs):
6     print('%d=%s' % (idx, item))
7
8 print('xs[1]=%s' % xs[1])
```

如果不需要下标，可以使用第 2 行的格式进行循环；如果需要下标，则用第 5 行的格式进行循环。

列表转化操作：例如列表中包含一组数字，若想得到一个新的列表，列表元素为原来列表元素的平方，又或者新列表只包含原列表中偶数的平方。这些操作都可以很方便地使用下面的列表转化操作（chp02/c007.py）：

```
1 x1 = [1, 2, 3, 4, 5, 6]
2 x2 = [x**2 for x in x1]
3 print('square:%s' % x2)
4 x3 = [x**2 for x in x1 if x % 2 == 0]
5 print('even square:%s' % x3)
```

程序运行结果：

```
square:[1, 4, 9, 16, 25, 36]
even square:[4, 16, 36]
```

第 2 行：x 为原始列表 x1 中的元素，并将元素的平方值作为新列表的元素。

第 4 行：x 为原始列表 x1 中的元素，将其中偶数元素的平方值作为新列表的元素。

3. 容器类之字典

Python 中的键值对的作用与 Java 中 Map 的作用类似。字典的常用操作如下（chp02/c008.py）：

```
1 #!/usr/bin/python
2 # vim: set fileencoding=utf-8 :
3
4 d = {'cat': '猫', 'dog': '狗', 'pig': '猪', 'cock': '公鸡', 'horse': '马'}
5 print('d[\'pig\']=%s' % d['pig'])
6 if 'dog' in d:
7     print('has dog')
8 else:
9     print('no dog')
10 d['donkey'] = '驴'
11 print('d:%s' % d)
12 print('monkey=%s; cock=%s' % (d.get('monkey', '...'), d.get('cock', 'no')))
13 print('d[\'cat\']=%s' % d.get('cat'))
14 del d['dog']
15 print('dog=%s' % d.get('dog'))
```

程序运行结果：

```
d['pig']=猪
has dog
d: {'cock': '公鸡', 'horse': '马', 'dog': '狗', 'donkey': '驴', 'pig': '猪', 'cat': '猫'}
monkey=...; cock=公鸡
d['cat']=None
dog=None
```

第 1、2 行：因为程序中用到了汉字常量，这里要告诉 Python 我们使用的字符集，否则在终端上运行程序会报错。

第 4 行：初始化字典对象。

第 5 行：打印某个键值内容，但是这里需要注意，该键值必须在字典中存在，否则会报异常，因此如果使用这个方法需要加异常处理逻辑。

第 6 行：判断某个键值是否存在于字典中。

第 10 行：为字典添加新键值对。

第 12 行：用字典对象的 `get` 方法获取键对应的值，如果该键不存在，可以返回第二个参数所指定的默认值。

第 13 行：用字典对象的 `get` 方法获取键对应的值，如果该键不存在，可以返回 `None` 而不会报错。

第 14 行：删除某个键值对。

字典元素遍历的方法如下：

```
1 d = {'cat': 2, 'dog': 3, 'pig': 5}
2 for item in d:
3     print('%s num:%d' % (item, d.get(item)))
4
5 for name, num in d.items():
6     print('%s has %d num' % (name, num))
```

程序运行结果：

```
pig num:5
dog num:3
cat num:2
pig has 5 num
dog has 3 num
cat has 2 num
```

第 2 行：遍历字典键。

第 5 行：同时遍历字典键和值。

下面利用转化操作生成乘方的表，代码如下：

```
1 xs = [0, 1, 2, 3, 4, 5, 6, 7, 8]
2 squares = {x: x**2 for x in xs if x%2 == 0}
3 print('squares:%s' % squares)
```

程序运行结果：

```
squares:{0: 0, 8: 64, 2: 4, 4: 16, 6: 36}
```

4. 容器类之集合

集合是元素没有顺序但不允许重复的容器类。在实际应用中，如果想取得不重复的元素，就可以将其赋给一个集合。

集合的基本操作如下：

```
1 xs = {'cat', 'dog', 'pig'}
2 print(('dog' in xs))
3 print(('tiger' in xs))
4 xs.add('fish')
5 print('len=%d' % len(xs))
6 xs.remove('cat')
7 print('xs:%s' % xs)
8 xs.remove('abc')
9 print('len=%d' % len(xs))
```

程序运行结果：

```
True
False
len=4
xs:set(['fish', 'dog', 'pig'])
Traceback (most recent call last):
  File "c011.py", line 8, in <module>
    xs.remove('abc')
KeyError: 'abc'
```

第 1 行：初始化集合变量。

第 2 行：判断元素是否在集合中，如果在则返回 **True**。

第 3 行：判断元素是否在集合中，如果不在则返回 **False**。

第 4 行：向集合中添加元素。

第 6 行：从集合中删除已有的元素。

第 8 行：从集合中删除没有的元素，这时会报异常。因此在实际使用时，需要先判断集合中是否存在该元素，然后再删除。

集合的遍历操作如下：

```
1 s1 = {'cat', 'dog', 'pig', 'horse', 'cock'}
2 for idx, item in enumerate(s1):
3     print('%d: %s' % (idx+1, item))
```

程序运行结果：

```
1: horse
2: cat
3: cock
4: dog
5: pig
```

注意：上面的程序在执行遍历操作时的顺序与添加的顺序不同。实际上，我们不能认为集合遍历是有顺序的，因为集合元素本身就是无序的。

5. 容器类之元组

Turple 元组是一个有序的列表，但是值不可变。元组与列表比较相似，只是元组可以作为字典的键，也可以作为集合的元素，而列表却不可以。

元组最常用的场合为返回值的情况，例如函数想返回多个值，又不想定义一个值对象，则可以用元组来返回。

元组的基本用法如下：

```
1 t1 = (2, 3, 4, 5, 6)
2 print('t1 type=%s' % type(t1))
3 print('t1[2]=%d' % t1[2])
4 for item in t1:
```

```

5     print('%s' % (item))
6 t2 = {(x, x+1): x for x in range(5)}
7 print('t2:%s' % t2)
8 t3 = (2, 3)
9 print('(2, 3)=%d' % t2[t3])

```

程序运行结果：

```

t1 type=<type 'tuple'>
t1[2]=4
2
3
4
5
6
t2:{(0, 1): 0, (1, 2): 1, (3, 4): 3, (2, 3): 2, (4, 5): 4}
(2, 3)=2

```

第 1 行：初始化一个元组。

第 2 行：打印元组的类型。

第 3 行：获取元组指定位置的值。

第 4、5 行：遍历元组。

第 6 行：将元组作为字典的键。

第 9 行：以元组为键，取出字典中相应的值。

6. 函数

Python 中使用 `def` 来定义函数，还可以使用默认参数，下面的程序是用快速排序函数定义的，代码如下（chp02/c014.py）：

```

1 def quicksort(arr, mode=1):
2     if len(arr) <= 1:
3         return arr
4     pivot = arr[len(arr) // 2]
5     if 1 == mode:
6         left = [x for x in arr if x < pivot]
7     else:
8         left = [x for x in arr if x > pivot]
9     middle = [x for x in arr if x == pivot]
10    if 1 == mode:
11        right = [x for x in arr if x > pivot]
12    else:
13        right = [x for x in arr if x < pivot]
14    return quicksort(left, mode) + middle + quicksort(right, mode)
15
16 if __name__ == '__main__':
17     a1 = [3, 6, 8, 10, 1, 2, 1]
18     print(quicksort(a1))
19     print(quicksort(a1, 2))

```

程序运行结果：

```

[1, 1, 2, 3, 6, 8, 10]
[10, 8, 6, 3, 2, 1, 1]

```

上面这段代码用到了递归程序设计的思想，所以排序方法看起来很简洁。具体说明如下。

第 1 行：定义快速排序函数，`arr` 为待排序列表，`mode` 默认值为 1 时代表递增排序，否则代表递减排序，返回值为排好序的数组。

第 2、3 行：定义递归函数的结束条件，即当数组元素只剩下一个时直接返回。

第 4 行：取出数组的中间元素。

下面以递增（mode=1）情况为例进行讲解。

第 5~8 行：将数组中小于中间元素的元素赋给 left 数组。

第 9 行：将等于中间元素的元素赋给 middle 数组。

第 10~13 行：将大于中间元素的元素赋给 right 数组。

第 14 行：对 left 递归调用本函数进行递增排序，对 right 递归调用本函数进行递增排序，并将 left+middle+right 作为结果返回。

7. 类

Python 是一门面向对象的程序设计语言，有很多面向对象的特性，例如多继承等。但是在本书中，我们主要使用 Python 进行科学计算，因此不必用到复杂的面向对象的机制，因为那样必然会降低程序的执行效率。但是类作为代码的一种组织方式，一起管理行为和数

据，对于我们来说，这还是比较方便的。

下面是我们的类定义代码：

```
1 class Greeter:
2     def __init__(self, name, email, pwd):
3         self.name = name
4         self._email = email
5         self.__pwd = pwd
6         self.__p1 = 'test private'
7
8     def greet(self, prefix):
9         print('%s %s' % (prefix, self.name))
10        print('pwd=%s' % self.__pwd)
11
12    def getP1(self):
13        return self.__p1
14
15 g = Greeter('u1', 'e1', '123')
16 g.greet('Hello')
17 print('name=%s' % g.name)
18 print('email=%s' % g._email)
19 print('p1=%s' % g.getP1())
20 print('pwd=%s' % g.__pwd)
```

程序运行结果：

```
Hello u1
pwd=123
name=u1
email=e1
p1=test private
Traceback (most recent call last):
  File "c015.py", line 20, in <module>
    print('pwd=%s' % g.__pwd)
AttributeError: Greeter instance has no attribute '__pwd'
```

第 1 行：定义 Greeter 类。

第 2 行：定义 Greeter 类构造函数，并且有 3 个参数。注意，函数定义虽然还是用 def，但是类方法的第一个参数应该是 self。

第 3 行：定义公共属性 name，根据惯例，普通变量名代表公共变量。

第 5、6 行：定义私有属性，即外部不能直接访问。

第 8 行：定义公共方法 greet，在其函数体内可以访问私有变量。

第 12 行：定义公共方法 getP1，这样在类的外部就可以访问__p1 的值了。

第 15 行：初始化一个类对象。

第 16 行：调用类的公共方法 `greet`，可以看到其可以获取私有变量 `pwd` 的值。

第 17 行：直接获取并打印公共属性的值。

第 19 行：通过公共方法获取私有属性的值。

第 20 行：直接获取私有属性的值并报错。

2.2.2 多维数组的使用

在简单介绍了 Python 语言的基本特性之后，下面来介绍一下 Python 的数值计算库 NumPy。可以说，正是由于 NumPy 库的存在，才使得 Python 在机器学习领域占据了重要位置。

首先来介绍 NumPy 中最重要的概念多维数组。多维数组是实际应用中频率最高的一种结构。下面先来看一下在 NumPy 中怎样初始化一个多维数组：

```
1 from numpy import *
2
3 arr1 = arange(15).reshape(3, 5)
4 print("Array:%s" % arr1)
5 print("Shape:%s" % str(arr1.shape))
6
```

在上面的程序中，第 1 行引入 NumPy 库，第 3 行声明了一个 3 行 5 列的数组，第 4 行打印这个数组，第 5 行打印数组的维度，结果如下：

```
Array:[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
Shape:(3, 5)
```

如果我们知道数组元素的值，想在数组初始化时直接赋值，则可以采用以下方式：

```
1 from numpy import *
2
3 arr1 = array([(1.0, 2.0, 3.0), (4.0, 5.0, 6.0)])
4 print("Array:%s" % arr1)
5 print("Shape:%s" % str(arr1.shape))
6
7 arr2 = array([[(1.0, 1.1, 1.2), (2.1, 2.2, 2.3)],
8               [(3.1, 3.2, 3.3), (4.1, 4.2, 4.3)],
9               [(5.1, 5.2, 5.3), (6.1, 6.2, 6.3)]]])
10 print("Array2:%s" % arr2)
11 print("Shape:%s" % str(arr2.shape))
12
13 arr3 = arange(18).reshape(3, 2, 3)
14 print("Array3:%s" % arr3)
15 print("Shape:%s" % str(arr3.shape))
```

在上面的代码中，分别初始化了 3 个数组。

第 3 行：因为知道数组元素初始值，初始化了一个 2×3 的二维数组。

第 7 行：在知道数组元素初始值的情况下，初始化一个 3×2×3 的三维数组，并打印数组内容和维度。

第 13 行：初始化一个三维数组，这时没有给定初始值，直接采用 `reshape` 的方式生成数组。

程序运行结果：

```
Array: [[ 1.  2.  3.]
 [ 4.  5.  6.]]
Shape: (2, 3)
Array2: [[[ 1.   1.1  1.2]
 [ 2.1  2.2  2.3]]

 [[ 3.1  3.2  3.3]
 [ 4.1  4.2  4.3]]

 [[ 5.1  5.2  5.3]
 [ 6.1  6.2  6.3]]]
Shape: (3, 2, 3)
Array3: [[[ 0  1  2]
 [ 3  4  5]]

 [[ 6  7  8]
 [ 9 10 11]]

 [[12 13 14]
 [15 16 17]]]
Shape: (3, 2, 3)
```

如果希望像其他语言一样生成一个数组，且数组元素的初始值全部为 0，则可以通过下面的代码来实现：

```
1 import numpy as np
2
3 arr = np.zeros((3, 5))
4 print(arr)
5
```

上面的代码将生成一个 3 行 5 列的二维数组，并且数组元素的初始值全部为 0。

如果希望为数组元素赋其他值，可以用如下方式（chp02/c016.py）：

```
1 import numpy as np
2
3 arr1 = np.ones((2, 3))
4 print(arr1)
5 arr2 = np.full((2, 3), 8.0)
6 print(arr2)
7 arr3 = np.eye(3)
8 print(arr3)
9 arr4 = np.random.random((2, 3))
10 print(arr4)
```

程序运行结果：

```
[[ 1.  1.  1.]
 [ 1.  1.  1.]]
[[ 8.  8.  8.]
 [ 8.  8.  8.]]
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
[[ 0.07397279  0.27584408  0.62033104]
 [ 0.98544126  0.25308855  0.93819575]]
```

第 3 行：初始化 2×3 数组，元素全为 1。

第 5 行：初始化 2×3 数组，元素值全为 8。

第 7 行：初始化 3×3 数组。

第 9 行：初始化 2×3 数组，元素值为随机数。

NumPy 的数组与 Python 中的列表类似，也支持灵活的切片操作，代码如下（chp02/c017.py）：

```

1 import numpy as np
2
3 a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
4 b = a[:2, 1:3]
5 print('b:%s' % b)
6 print('a[0, 1]=%d' % a[0, 1])
7 b[0, 0] = 88
8 print('a[0, 1]=%d, b[0, 0]=%d' % (a[0, 1], b[0, 0]))

```

程序运行结果:

```

b: [[2 3]
     [6 7]]
a[0, 1]=2
a[0, 1]=88, b[0, 0]=88

```

第3行: 定义一个 3×4 数组。

第4行: 定义数组 b, 取数组 a 第1维 0、1 两个元素, 以及第2维 1、2 两个元素。

第7行: 改变[0,0]位置元素的值。

第8行: 打印 b[0,0]和 a[0,1]的值, 由于它们指向同一个位置, 所以它们的值均为更改以后的值。

NumPy 对多维数组切片操作可以起到降维的作用, 代码如下 (chp02/c018.py):

```

1 import numpy as np
2
3 a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
4 r1 = a[1:2, :]
5 print('%s:%s' % (r1.shape, r1))
6 r2 = a[1, :]
7 print('%s:%s' % (r2.shape, r2))
8 c1 = a[:, 1:2]
9 print('%s:%s' % (c1.shape, c1))
10 c2 = a[:, 1]
11 print('%s:%s' % (c2.shape, c2))

```

程序运行结果:

```

(1, 4): [[5 6 7 8]]
(4,): [5 6 7 8]
(3, 1): [[ 2]
          [ 6]
          [10]]
(3,): [ 2  6 10]

```

第3行: 生成 3×4 的 NumPy 多维数组。

第4、5行: 取出数组 a 的第2行, 并返回二维数组。

第6、7行: 取出数组 a 的第2行, 作为一维数组返回。

第8、9行: 取出数组 a 的第2列, 作为二维数组返回。

第10、11行: 取出数组 a 的第2列, 作为一维数组返回。

NumPy 多维数组可以使用整数数组索引, 这样非常方便修改数组元素中的值, 例如将指定位置数组元素加上一个数等操作, 代码如下 (chp02/c019.py):

```

1 import numpy as np
2
3 a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
4 print('a:%s' % a)
5 a1 = a[[0, 1, 2], [0, 1, 0]]
6 print('a1:%s' % a1)
7 a2 = np.array([a[0, 0], a[1, 1], a[2, 0]])
8 print('a2:%s' % a2)
9 a3 = a[[0, 0], [1, 1]]
10 print('a3:%s' % a3)
11 a4 = np.array([a[0, 1], a[0, 1]])

```

```

12 print('a4:%s' % a4)
13
14 b = np.array([0, 2, 0])
15 a5 = a[range(3), b]
16 print('a5:%s' % a5)
17 a6 = a[range(3), b] + 10
18 print('a:%s' % a)
19 print('a6:%s' % a6)
20 a[range(3), b] += 10
21 print('new:%s' % a)

```

程序运行结果：

```

a: [[ 1  2  3  4]
     [ 5  6  7  8]
     [ 9 10 11 12]]
a1: [1 6 9]
a2: [1 6 9]
a3: [2 2]
a4: [2 2]
a5: [1 7 9]
a: [[ 1  2  3  4]
     [ 5  6  7  8]
     [ 9 10 11 12]]
a6: [11 17 19]
new: [[11  2  3  4]
       [ 5  6 17  8]
       [19 10 11 12]]

```

第 3 行：生成 3×4 的数组。

第 5 行：采用整数数组索引，取出一维子数组，即取出[0, 0][1, 1][2, 0]元素组成一维数组。

第 7 行：第 5 行的等价形式。

第 9 行：与第 5 行类似，只是取出同一位置的元素组成数组。

第 11 行：第 9 行的等价形式。

第 14 行：定义整数数组 b。

第 15 行：相当于整数数组索引[0, 1, 2][0, 2, 0]。

第 17 行：数组加 10，等于数组中每个元素加 10。

第 18、19 行：整数数组索引取出的数组与原数组是两个完全独立的数组。

第 20 行：通过整数数组索引，指定特定的数组元素，在这些元素上统一加 10。

在 NumPy 多维数组中，还可以使用布尔表达式作为下标来选择元素，代码如下（chp02/c020.py）：

```

1 import numpy as np
2
3 a = np.array([[1, 2], [3, 4], [5, 6]])
4 b = (a > 2)
5 print('b.type:%s\r\n%s' % (type(b), b))
6 c = a[b]
7 print('c.type:%s\r\n%s' % (type(c), c))
8 d = a[a>2]
9 print('d.type:%s\r\n%s' % (type(d), d))

```

程序运行结果：

```

b.type:<class 'numpy.ndarray'>
[[False False]
 [ True  True]
 [ True  True]]
c.type:<class 'numpy.ndarray'>
[3 4 5 6]
d.type:<class 'numpy.ndarray'>
[3 4 5 6]

```

第 3 行：定义 3×2 多维数组。

第 4 行：判断数组元素值是否大于 2，并将结果形成新的 3×2 多维数组。

第 6 行：取出元素值大于 2 的数组元素，组成一个一维数组返回。

第 8 行：取出数组中大于 2 的所有元素，组成一个一维数组返回。

与 Python 中的列表不同，NumPy 数组只能保存同一类型的元素，但 NumPy 可以自动判断数组元素的类型。在初始化数组时，也可以显式地指定数组类型，代码如下（chp02/c021.py）：

```
1 import numpy as np
2
3 a = np.array([[1, 2], [3, 4], [5, 6]])
4 print('a shape:%s, type:%s' % (a.shape, a.dtype))
5 b = np.array([[1.0, 2.0], [3.0, 4.0]])
6 print('b shape:%s, type:%s' % (b.shape, b.dtype))
7 c = np.array([[1.0, 2.0], [3.0, 4.0]], dtype=np.float32)
8 print('c shape:%s, type:%s' % (c.shape, c.dtype))
```

程序运行结果：

```
a shape:(3, 2), type:int64
b shape:(2, 2), type:float64
c shape:(2, 2), type:float32
```

第 3 行：生成数组 a，在没有指定类型时，NumPy 默认是 int64 类型的。

第 5 行：利用浮点数初始化数组 b，在没有指定类型时，NumPy 默认是 float64 类型的。

第 7 行：利用浮点数初始化数组 c，并指定类型为 float32。

NumPy 数组可以进行加、减、乘、除操作，即对相应的元素进行加、减、乘、除操作。注意：此处的乘法与向量与向量、向量与矩阵、矩阵与矩阵间的乘法是不同的，代码如下（chp02/c022.py）：

```
1 import numpy as np
2
3 a1 = np.array([[1, 2], [3, 4]], dtype=np.float64)
4 a2 = np.array([[5, 6], [7, 8]], dtype=np.float64)
5 print('a1+a2=%s' % (a1+a2))
6 print('np.add:%s' % (np.add(a1, a2)))
7 print('a2-a1=%s' % (a2 - a1))
8 print('np.subtract:%s' % (np.subtract(a2, a1)))
9 print('a1*a2=%s' % (a1 * a2))
10 print('np.multiply:%s' % (np.multiply(a1, a2)))
11 print('a1/a2=%s' % (a1 / a2))
12 print('np.divide:%s' % (np.divide(a1, a2)))
```

程序运行结果：

```
a1+a2=[[ 6.  8.]
 [10. 12.]]
np.add:[[ 6.  8.]
 [10. 12.]]
a2-a1=[[ 4.  4.]
 [ 4.  4.]]
np.subtract:[[ 4.  4.]
 [ 4.  4.]]
a1*a2=[[ 5. 12.]
 [21. 32.]]
np.multiply:[[ 5. 12.]
 [21. 32.]]
a1/a2=[[ 0.2          0.33333333]
 [ 0.42857143  0.5       ]]
np.divide:[[ 0.2          0.33333333]
 [ 0.42857143  0.5       ]]
```

第 3 行：生成 2×2 数组 a1，并指定类型为 `numpy.float64`。

第 4 行：生成 2×2 数组 a2，并指定类型为 `numpy.float64`。

第 5 行：`a1+a2` 表示数组对应元素相加得到的新数组。

第 6 行：使用 `np.add` 函数，效果与第 5 行相同。

第 7 行：`a2-a1` 表示数组对应元素相减所得到的新数组。

第 8 行：使用 `np.subtract` 函数，效果与第 7 行相同。

第 9 行：用 `a1×a2` 表示数组对应元素相乘所得到的新数组。

第 10 行：使用 `np.multiply` 函数，效果与第 9 行相同。

第 11 行：`a1/a2` 表示数组对应元素相除所得到的新数组。

第 12 行：使用 `np.divide` 函数，效果与第 11 行相同。

关于 NumPy 向量运算和矩阵运算的详细说明，读者可以参考 2.2.3 节和 2.2.4 节的内容。

这里只简单介绍一下 NumPy 的点积操作。

向量和矩阵的点积操作的代码如下（`chp02/c023.py`）：

```
1 import numpy as np
2
3 v1 = np.array([9, 10])
4 v2 = np.array([11, 12])
5 print('vector dot: v1*v2=%s' % v1.dot(v2))
6 print('vector dot: v1*v2=%s' % np.dot(v1, v2))
7
8 m1 = np.array([[1, 2], [3, 4]])
9 m2 = np.array([[5, 6], [7, 8]])
10 print('vector product matrix: m1*v1=%s' % m1.dot(v1))
11 print('vector product matrix: m1*v1=%s' % np.dot(m1, v1))
12 print('matrix product matrix: m1*m2=\r\n%s' % m1.dot(m2))
13 print('matrix product matrix: m1*m2=\r\n%s' % np.dot(m1, m2))
```

程序运行结果：

```
vector dot: v1*v2=219
vector dot: v1*v2=219
vector product matrix: m1*v1=[29 67]
vector product matrix: m1*v1=[29 67]
matrix product matrix: m1*m2=
[[19 22]
 [43 50]]
matrix product matrix: m1*m2=
[[19 22]
 [43 50]]
```

第 3、4 行：声明两个向量，在 NumPy 中用一维数组代表向量。

第 5、6 行：用两种不同的方式求向量点积，结果是相同的。

第 8、9 行：声明两个 2×2 矩阵，在 NumPy 中用二维数组代表矩阵。

第 10、11 行：用两种不同的方式实现矩阵与向量的点积，结果是相同的。

第 12、13 行：用两种不同的方式实现矩阵与矩阵的点积，结果是相同的。

在矩阵指定维度上求和的代码如下（`chp02/c024.py`）：

```
1 import numpy as np
2
3 W = np.array([[1, 2], [3, 4]])
4 print('sum(W)=%d' % np.sum(W))
5 print('sum_col=%s' % np.sum(W, axis=0))
6 print('sum_row=%s' % np.sum(W, axis=1))
```

程序运行结果：

```
sum(W)=10
sum_col=[4 6]
sum_row=[3 7]
```

第3行：声明一个 2×2 矩阵。

第4行：求出矩阵所有元素之和。

第5行：求出矩阵每列之和，并以一维数组形式返回。

第6行：求出矩阵每行之和，并以一维数组形式返回。

向量和矩阵的转置操作的代码如下（chp02/c025.py）：

```
1 import numpy as np
2
3 x = np.array([1, 2, 3])
4 print('x:\r\n%s' % x)
5 print('x.T:\r\n%s' % x.T)
6 W = np.array([[1, 2], [3, 4]])
7 print('W:\r\n%s' % W)
8 print('W.T:\r\n%s' % W.T)
```

程序运行结果：

```
x:
[1 2 3]
x.T:
[1 2 3]
W:
[[1 2]
 [3 4]]
W.T:
[[1 3]
 [2 4]]
```

第3行：声明 \mathbb{R}^3 向量，此向量也可视为 $\mathbb{R}^{3 \times 1}$ 矩阵。

第5行：打印向量的转置，其实际上变为 $\mathbb{R}^{1 \times 3}$ 矩阵，但是由于 NumPy 中均用一维数组表示，因此打印时看不出区别。

第6行：声明 $\mathbb{R}^{2 \times 2}$ 矩阵。

第8行：打印矩阵的转置，即行和列进行了互换。

下面介绍 NumPy 的广播操作。

NumPy 的广播操作可以对不同维度的数组进行算术运算，NumPy 自动扩展数组维度进行高效运算。下面先来看一下，如果不使用 NumPy 的广播操作特性，怎样来实现这些功能，然后再看看在 NumPy 的广播操作下如何实现这些功能，最后给出几个常用的实例。

下面来看看用普通的方法，怎样将一个向量加到一个矩阵的每一行中，代码如下（chp02/c026.py）：

```
1 import numpy as np
2
3 W = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
4 x = np.array([1, 0, 1])
5 y = np.empty_like(W)
6 # method 1
7 for i in range(4):
8     y[i, :] = W[i, :] + x
9 print('y=%s' % y)
10
11 # method 2
12 v1 = np.tile(x, (4, 1))
13 print('v1=%s' % v1)
14 y1 = W + v1
15 print('y1=%s' % y1)
```

程序运行结果：

```
y=[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
v1=[[1 0 1]
 [1 0 1]
 [1 0 1]
 [1 0 1]]
y1=[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

第 3 行：声明一个 $\mathbb{R}^{4 \times 3}$ 矩阵 W 。

第 4 行：声明一个 \mathbb{R}^3 向量。

第 5 行：生成一个空的与矩阵 W 维度相同的矩阵。

第 7、8 行：令矩阵 y 的每一行等于矩阵 W 对应行与向量 x 的转置相加。

第 9 行：打印 y 的值。

第 12 行：生成一个 4 行 1 列由列向量 x 组成的矩阵，其实际维度为 $\mathbb{R}^{4 \times 3}$ 。

第 14 行：矩阵 $y1$ 定义为矩阵 W 和 $v1$ 的和，即对应位置元素之和组成的矩阵。

可以看出，这两种方法实现的功能相同，但是第一种方法由于需要循环操作，所以效率会稍差一些。NumPy 对第二种方法提供了内置支持，我们称之为广播操作，就是将不同维的数组操作变成同维的数组操作，代码如下（chp02/c027.py）：

```
1 import numpy as np
2
3 W = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
4 x = np.array([1, 0, 1])
5 y = W + x
6 print('y=%s' % y)
```

程序运行结果：

```
y=[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

第 3 行：声明一个 $\mathbb{R}^{4 \times 3}$ 矩阵 W 。

第 4 行：声明一个 \mathbb{R}^3 向量。

第 5 行：声明矩阵 y 为矩阵 W 加上向量 x ，这时 NumPy 会自动利用向量 x 生成 $\mathbb{R}^{4 \times 3}$ 矩阵并将其与矩阵 W 相加。与前面两种方法相比，这种方法不仅在表达上更简洁，而且在实际执行中效率也是最高的。

2.2.3 向量运算

在深度学习算法中，向量是非常重要的类型，比如神经网络的输入通常可以表示为一个输入向量，神经网络的输出也可以表示为一个向量，因此我们有必要研究一下向量在 NumPy 中的用法。

在数学中，向量 V 是一个由 M 数组成的数列，用 M 行 1 列的形式进行存放，由于不便于用书面方式表示，故通常用一个一维数组的转置来表示向量，例如：

$$\begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_m \end{bmatrix} \text{ 等价于 } [v_1 \ v_2 \ \dots \ v_m]^T$$

因此，一维数组可以视为矩阵转置后的表示，所以针对矩阵的操作就演变为针对一维数组的操作。下面来看一个简单的矩阵相加的例子。可能有些读者会有疑问，为什么我们非要使用 NumPy 来表示数组或矩阵呢？直接用 Python 原生的列表不是很方便吗？即使语法上稍显复杂，也没有比 NumPy 差很多呀！在下面的矩阵相加的例子中，将分别演示使用 NumPy 库和原生 Python 的程序，其中记录了程序运行的时间，经过对比，相信读者就会发现为什么要花这么大精力来安装和使用 NumPy 库了。

采用 NumPy 库的矩阵相加的代码如下：

```
1 import numpy as np
2 import sys
3 import datetime as dt
4
5 def main():
6     n = 200
7     startTime = dt.datetime.now()
8     a = np.arange(n)**2
9     b = np.arange(n)**3
10    c = a + b
11    endTime = dt.datetime.now()
12    runTime = endTime - startTime
13    print(c)
14    print("execute time=%d" % runTime.microseconds)
15
16 if __name__ == '__main__':
17     main()
18
```

在上面的代码中，第 8 行声明了一个有 200 个元素的一维数组，每个元素是其下标值的平方。第 9 行也声明了一个有 200 个元素的一维数组，每个元素是其下标值的立方。

在第 7 行，程序开始执行之前，首先记录系统的时间戳，然后进行矩阵加法运算，再打印运算结果，最后打印程序的运行时间。

采用原生 Python 的程序代码如下：

```
1 import sys
2 import datetime as dt
3
4 def main():
5     n = 200
6     startTime = dt.datetime.now()
7     a = range(n)
8     b = range(n)
9     c = []
10    for i in range(len(a)):
11        a[i] = i**2
12        b[i] = i**3
13        c.append(a[i] + b[i])
14    endTime = dt.datetime.now()
15    runTime = endTime - startTime
16    print(c)
17    print("execute time=%d" % runTime.microseconds)
18
19 if __name__ == '__main__':
20     main()

```


在笔者的机器上，采用的是 VMware 虚拟机，两个程序的运行时间相差 100ms 左右。若在深度学习算法中的神经网络训练阶段，运算量将数以亿计，这时两者之间性能的差异就会更明显地表现出来，所以采用 NumPy 是非常有必要的选择。

向量的乘法是两个向量的点积，代码如下：

```
1 import numpy as np
2
3 v1 = np.arange(5)
4 v2 = np.arange(5)**2
5 v3 = np.dot(v1, v2)
6
7 print(v1)
8 print(v2)
9 print(v3)
```

程序首先在第 3、4 行生成两个一维数组作为向量，然后调用 NumPy 库的点积函数 dot 求出这两个向量的点积结果。程序运行的结果为 100，表明进行的确实是点积运算。

2.2.4 矩阵运算

由于矩阵应用得十分广泛，而且向量可以视为列为 1 的矩阵，所以 NumPy 中有专门的矩阵类，均继承自 ndarray，且具有数组的属性和方法。

1. 矩阵初始化

在 NumPy 库中，函数 mat、matrix、bmat 均可以用于创建矩阵。对于已经知道矩阵初始值的情况，可以通过矩阵初始化字符串生成对应的矩阵。

下面来看一下矩阵初始化过程要用到的初始化字符串格式，其实很简单，就是以行为单位，行与行之间用分号分隔，行内数据用空格分隔，例如要初始化如下矩阵：

$$\begin{bmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \\ 7.0 & 8.0 & 9.0 \end{bmatrix}$$

其所对应的初始化字符串可以表示为：

```
"1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0"
```

代码如下：

```
1 import numpy as np
2
3 A = np.mat('1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0')
4 print(A)
5
6
```

运行结果：

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]
 [ 7.  8.  9.]]
```

读者也许已经注意到，其实用矩阵类能实现的操作用多维数组完全也可以实现，因此

在科学计算领域，专家们更推荐直接使用多维数组 `ndarray` 来做矩阵运算。所以在下面的例子中，我们将采用 `ndarray` 来进行各种矩阵运算。

2. 矩阵加减法运算

矩阵加减法运算要求两个矩阵维度相同，其结果是对应位置元素依次相加或相减。采用 `ndarray` 的实现代码如下：

```
1 import numpy as np
2
3 A = np.array([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
4 B = np.array([[1.0, 1.0], [1.0, 1.0], [1.0, 1.0]])
5 C = A + B
6 print("A+B=\r\n%s" % C)
7 D = A - B
8 print("A-B=\r\n%s" % D)
9
```

上面的代码先分别定义了两个 3×2 矩阵，然后依次求出矩阵相加和相减的结果，并打印出结果。

3. 标量与矩阵相乘

根据数学知识我们知道，数与矩阵的乘法，是将数依次乘以矩阵中的每一个元素，代码如下：

```
1 import numpy as np
2
3 A = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
4 B = 2 * A
5 print("2*A=\r\n%s" % B)
6
```

通过打印结果可以看到，“ $2 \times A$ ”的结果是矩阵 **A** 的元素均乘以 2 得到的。

4. 矩阵乘法

矩阵与常数相乘的代码实现如下：

```
1 import numpy as np
2
3 A = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
4 B = 2 * A
5 print("2*A=\r\n%s" % B)
6
```

上面的代码会将矩阵 **A** 的每一个元素的值都扩大两倍。

根据线性代数中的知识，我们知道 **A** 和 **B** 两个矩阵相乘，则要求矩阵 **A** 的列数与矩阵 **B** 的行数相同，对于乘积结果矩阵 $C=A \times B$ ，则 **C** 的每一个元素可以由下面的公式来计算。

假设 $A_{m \times p}$ 且 $B_{p \times n}$ ，则：

$$c_{i,j} = \sum_{k=1}^p a_{i,p} \times b_{p,j}$$

具体实现代码如下：

```
1 import numpy as np
2
3 A = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
4 B = np.array([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
5 C = np.dot(A, B)
6 print(C)
7
```

这里需要注意的是，“ $A \times B$ ”并不是通常的矩阵乘法，而是矩阵的点积操作，是矩阵中对应位置元素相乘，因此要求矩阵 A 和 B 的维度相同，最后的结果矩阵的维度仍然是原来矩阵的维度。

2.2.5 线性代数

1. 矩阵的转置

矩阵的转置为将矩阵的行变为列，将列变为行，代码如下：

```
1 import numpy as np
2
3 A = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
4 print("Transpose:\r\n%s" % A.T)
5
```

2. 矩阵求逆

对于矩阵 A ，我们定义 $AA^{-1} = I$ ，其中 I 为单位矩阵，其主对角线元素均为 1，其余元素均为 0，例如：

$$AA^{-1} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{bmatrix} \begin{bmatrix} -1.48 & 0.36 & 0.88 \\ 0.56 & 0.08 & -0.36 \\ 0.16 & -0.12 & 0.04 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

求矩阵的逆矩阵可以使用 `scipy.linalg.inv`，代码如下：

```
1 import numpy as np
2 from scipy import linalg
3
4 A = np.array([[1, 3, 5], [2, 5, 1], [2, 3, 8]])
5 A_1 = linalg.inv(A)
6 print("A_1=\r\n%s" % A_1)
7
```

注意：在上面的程序中，我们使用了 Python 科学计算库 SciPy，在后面用到线性代数计算的地方，我们都会使用 SciPy 库。

3. 方阵行列式的值

方阵 A （行和列相等）的行列式用 $|A|$ 来表示，我们用 $a_{i,j}$ 来表示矩阵 A 第 i 行第 j 列元素，并且定义 $M_{i,j}=|A_{i,j}|$ ，即去掉方阵 A 的第 i 行和第 j 列后形成的新矩阵，则方阵 A 的行列式的值可以定义为：

$$|A| = \sum_{j=0}^n (-1)^{i+j} a_{i,j} M_{i,j}$$

从上面的定义可以看出，方阵行列式的值是通过递归来定义的。只说概念比较抽象，下面以一个具体的方阵为例，实际演示一下行列式的值的求法：

$$A = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 5 & 1 \\ 2 & 3 & 8 \end{bmatrix}$$

对第1行应用上述公式，可以得到如下结果：

$$\begin{aligned} |A| &= (-1)^{0+0} 1 \times \begin{vmatrix} 5 & 1 \\ 3 & 8 \end{vmatrix} + (-1)^{0+1} 3 \times \begin{vmatrix} 2 & 1 \\ 2 & 8 \end{vmatrix} + (-1)^{0+2} 5 \times \begin{vmatrix} 2 & 5 \\ 2 & 3 \end{vmatrix} \\ &= 1 \times \begin{vmatrix} 5 & 1 \\ 3 & 8 \end{vmatrix} - 3 \times \begin{vmatrix} 2 & 1 \\ 2 & 8 \end{vmatrix} + 5 \times \begin{vmatrix} 2 & 5 \\ 2 & 3 \end{vmatrix} \\ &= 1 \times (5 \times 8 - 1 \times 3) - 3 \times (2 \times 8 - 1 \times 2) + 5 \times (2 \times 3 - 5 \times 2) = -25 \end{aligned}$$

在 SciPy 库中，方阵行列式的值可以由 det 求出，具体代码如下：

```
1 import numpy as np
2 from scipy import linalg
3
4 A = np.array([[1, 3, 5], [2, 5, 1], [2, 3, 8]])
5 print("A's determinant:\r\n%f" % (linalg.det(A)))
6
```

读者可以看到，虽然方阵行列式的值的定义比较复杂，但是利用 SciPy 库进行计算，程序实现起来还是比较简单的。

4. 矩阵的范数

矩阵和向量都可以定义范数，范数具有不同的阶，而在实际应用中，一阶和二阶比较常见，下面将进行重点介绍。

对于向量而言，一阶范数定义为：

$$|v| = \sum_{i=0}^M |v_i|$$

二阶范数定义为：

$$|v| = \left(\sum_{i=0}^M v_i^2 \right)^{\frac{1}{2}}$$

即向量 v 的长度。

对于矩阵而言，一阶范数定义为：

$$\|A\| = \max_j \sum_{i=0}^M |a_{i,j}|$$

二阶范数定义为：

$$\|A\|^2 = \max \sigma_i$$

式中， σ_i 是矩阵 A 的奇异值。

在实际应用中，矩阵 A 还有一个范数经常会被用到，定义如下：

$$\|A\|^{\text{fro}} = \sqrt{\text{trace}(A^H A)}$$

代码如下：

```
1 import numpy as np
2 from scipy import linalg
3
4 v1 = np.array([1, 2, 3])
5 print("v1 norm_1:\r\n%s" % linalg.norm(v1, 1))
6 print("v1 norm_2:\r\n%s" % linalg.norm(v1, 2))
7
8 A = np.array([[1, 2], [3, 4]])
9 print("matrix norm_1:\r\n%s" % linalg.norm(A, 1))
10 print("matrix norm_2:\r\n%s" % linalg.norm(A, 2))
11 print("matrix norm_fro:\r\n%s" % linalg.norm(A, 'fro'))
12
```

计算结果为：

```
v1 norm_1:
6.0
v1 norm_2:
3.74165738677
matrix norm_1:
6.0
matrix norm_2:
5.46498570422
matrix norm_fro:
5.47722557505
```

5. 线性最小二乘法

线性最小二乘法也许是最简单的神经网络学习算法，在应用数学的许多领域都有广泛的应用。在这类问题中，数据 y_i 是由输入数据 x_i 产生的，可以表示为由输入数据 x_i 通过协系数集 c_j 和模型函数 $f_j(x_i)$ 经过运算得到。可以表示为如下形式：

$$y_i = \sum_{j=1}^N c_j f_j(x_i) + \epsilon_i$$

式中， ϵ_i 代表模型的不确定性。线性最小二乘法的策略是通过找到协系数集，使通过模型计算得到的值与实际值误差达到最小，其误差函数可以定义为：

$$J(c) = \sum_{i=1}^N \left| y_i - \sum_{j=1}^Z c_j f_j(x_i) \right|^2$$

式中， N 代表总样本数， Z 代表模型的阶数。

现在的任务就变为通过调整协系数，使误差函数值达到最小。根据数学理论可知，可以通过对误差函数 $J(c)$ 的各个协系数求偏导，令偏导值为 0 来达到我们的目的：

$$\frac{\partial J}{\partial c_j} = 2 \sum_{i=1}^N \left(y_i - \sum_{j=1}^Z c_j f_j(x_i) \right) (-f_j(x_i)) = 0$$

在上式中， $j=1, 2, \dots, Z$ ，可以从式中解出 c_j ，从而达到我们的目的。

将上式重新进行组合，则：

$$\sum_{i=1}^N f_j(x_i) \sum_{j=1}^Z c_j f_j(x_i) = \sum_{i=1}^N y_i f_j(x_i) \quad (1)$$

对式 (1) 而言，我们假设共有 3 个样本，即 $i=1,2,3$ ，协系数的阶数为 2，即 $j=1,2$ ，则对不同的 j ，展开等式左边可以得到以下结果。

$j=1$:

$$f_1(x_1)[c_1 f_1(x_1) + c_2 f_2(x_1)] + f_1(x_2)[c_1 f_1(x_2) + c_2 f_2(x_2)] + f_1(x_3)[c_1 f_1(x_3) + c_2 f_2(x_3)] \quad (1.1)$$

$j=2$:

$$f_2(x_1)[c_1 f_1(x_1) + c_2 f_2(x_1)] + f_2(x_2)[c_1 f_1(x_2) + c_2 f_2(x_2)] + f_2(x_3)[c_1 f_1(x_3) + c_2 f_2(x_3)] \quad (1.2)$$

将式 (1.1) 按照 c_1 和 c_2 进行因式分解，可以得到如下公式：

$$[f_1(x_1)f_1(x_1) + f_1(x_2)f_1(x_2) + f_1(x_3)f_1(x_3)]c_1 + [f_1(x_1)f_2(x_1) + f_1(x_2)f_2(x_2) + f_1(x_3)f_2(x_3)]c_2 \quad (1.3)$$

将式 (1.2) 按照 c_1 和 c_2 进行因式分解并进行整理可得：

$$[f_2(x_1)f_1(x_1) + f_2(x_2)f_1(x_2) + f_2(x_3)f_1(x_3)]c_1 + [f_2(x_1)f_2(x_1) + f_2(x_2)f_2(x_2) + f_2(x_3)f_2(x_3)]c_2 \quad (1.4)$$

综合式 (1.3) 和式 (1.4) 可以得到矩阵，形式如下：

$$\begin{bmatrix} f_1(x_1) & f_1(x_2) & f_1(x_3) \\ f_2(x_1) & f_2(x_2) & f_2(x_3) \end{bmatrix} \begin{bmatrix} f_1(x_1) & f_2(x_1) \\ f_1(x_2) & f_2(x_2) \\ f_1(x_3) & f_2(x_3) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$$

设上式中间项为矩阵 \mathbf{A} ，以小写粗体字母代表向量，则上式可以表示为：

$$\mathbf{A}^T \mathbf{A} \mathbf{c}$$

同理，可以按不同的 j 值写出具体的式 (1) 右边的表达式。

$j=1$:

$$y_1 f_1(x_1) + y_2 f_1(x_2) + y_3 f_1(x_3) \quad (1.5)$$

$j=2$:

$$y_1 f_2(x_1) + y_2 f_2(x_2) + y_3 f_2(x_3) \quad (1.6)$$

将式（1.5）和式（1.6）组合成向量可以表示为如下矩阵相乘的形式：

$$\begin{bmatrix} f_1(x_1) & f_1(x_2) & f_1(x_3) \\ f_2(x_1) & f_2(x_2) & f_2(x_3) \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

按照小写粗体字母代表向量的定义，可以表示为：

$$\mathbf{A}^T \mathbf{y}$$

由等式左边与右边相等，可以得到：

$$\mathbf{A}^T \mathbf{A} \mathbf{c} = \mathbf{A}^T \mathbf{y}$$

即解出协系数 \mathbf{c} ：

$$\mathbf{c} = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{y} = \mathbf{A}^\dagger \mathbf{y}$$

式中， \mathbf{A}^\dagger 是矩阵 \mathbf{A} 的伪逆，则我们的模型可以表示为：

$$\mathbf{y} = \mathbf{A} \mathbf{c} + \epsilon$$

上述模型可以用 `scipy.linalg.lstsq` 函数求解，同时矩阵 \mathbf{A} 的伪逆 \mathbf{A}^\dagger 可以由 `scipy.linalg.pinv` 或 `scipy.linalg.pinv2` 函数得到，这两个函数得出的结果相同，只是具体算法不同。

下面以一个具体的实例来讲述详细的求解过程。假设我们有 10 个样本， $x_i=0.1 \times i$ ，而要预测的模型为：

$$y_i = c_1 e^{-x_i} + c_2 x_i \quad (2)$$

式中， $c_1=5.0$ 、 $c_2=2.0$ 。计算值为真实值，而在实际中所观测到的结果会存在误差。假设我们观测到的结果为 z_i ，其值在 y_i 的基础上加入随机噪声，代码如下：

```
1 import numpy as np
2 from scipy import linalg
3 #import matplotlib.pyplot as plt
4
5 c1, c2 = 5.0, 2.0
6 i = np.arange(1,11)
7 xi = 0.1*i
8 yi = c1*np.exp(-xi) + c2*xi
9 zi = yi + 0.05 * np.max(yi) * np.random.randn(len(yi))
10
11 A = np.c_[np.exp(-xi), xi], xi[:, np.newaxis]]
12 print("A:%s\r\n" % A)
13
14 c, resid, rank, sigma = linalg.lstsq(A, zi)
15 print("c:%s\r\n" % c)
16 print("resid:%s\r\n" % resid)
17 print("rank:%s\r\n" % rank)
18 print("sigma:%s\r\n" % sigma)
19
```

第5~9行生成我们观测到的数据,在式(2)基础上加入了随机噪声,形成观测到的 z_i ,用于下面的线性最小二乘法算法。

第11行定义了矩阵 A ,第14行调用`scipy.linalg.lstsq`函数求出协系数的值,并打印出来,结果如下:

```
A: [[ 0.90483742  0.1      ]
      [ 0.81873075  0.2      ]
      [ 0.74081822  0.3      ]
      [ 0.67032005  0.4      ]
      [ 0.60653066  0.5      ]
      [ 0.54881164  0.6      ]
      [ 0.4965853   0.7      ]
      [ 0.44932896  0.8      ]
      [ 0.40656966  0.9      ]
      [ 0.36787944  1.       ]]

c: [ 4.93848392  2.05630661]

resid:0.633212080784

rank:2

sigma: [ 2.58763467  1.02933937]
```

可以看出,程序比较准确地计算出了协系数值,达到了预期的效果。

6. 矩阵特征值和特征向量分解

方阵的特征值和特征向量分解,是实际中应用最广泛的一种形式。其定义为对于 n 阶矩阵 A ,如果数 λ 和 n 维非0列向量 x 使关系式 $Ax = \lambda x$ 成立,那么数 λ 称为矩阵 A 的特征值,向量 x 称为矩阵 A 对应于特征值 λ 的特征向量。

在SciPy中,求矩阵 A 的特征值和特征向量比较简单,但是需要注意的是,SciPy所求出的特征值是复数,由于给定的矩阵存在实数特征值,所以只显示其实数部分,代码如下:

```
1 import numpy as np
2 from scipy import linalg
3
4 A = np.array([[1, 5, 2], [2, 4, 1], [3, 6, 2]])
5 la, v = linalg.eig(A)
6 l1, l2, l3 = la
7 v1 = v[:, 0]
8 v2 = v[:, 1]
9 v3 = v[:, 2]
10
11 print("%s:%s" % (l1.real, v1))
12 print("%s:%s" % (l2.real, v2))
13 print("%s:%s" % (l3.real, v3))
14
```

运行该程序得到的结果为:

```
7.95791620491: [-0.5297175 -0.44941741 -0.71932146]
-1.25766470568: [-0.90730751  0.28662547  0.30763439]
0.299748500767: [ 0.28380519 -0.39012063  0.87593408]
```

即得到了正确的结果。

7. 矩阵的奇异值分解

根据特征值和特征向量的定义,我们知道只有方阵才有特征值和特征向量分解。对于普通的 $M \times N$ 矩阵,为了研究其性质,引入了奇异值分解。对于任意 $M \times N$ 矩阵 A , $A^T A$ 为

$N \times N$ 的厄米特方阵，而 AA^T 为 $M \times M$ 的厄米特方阵。厄米特方阵具有实数且非负的特征值，并且 $A^T A$ 及 AA^T 最多有 $\min(M, N)$ 个非 0 相同的特征值，我们将这些特征值定义为 σ_i^2 ，则 σ_i 为矩阵 A 的奇异值。

定义 $V = A^T A$ 、 $U = AA^T$ ， Σ 为 $M \times N$ 的对角阵，元素为矩阵 A 的奇异值，则矩阵 A 的奇异值分解可以表示为：

$$A = U \Sigma V^T$$

利用 SciPy 求矩阵 A 的奇异值分解的代码如下：

```

1 import numpy as np
2 from scipy import linalg
3
4 A = np.array([[1, 2, 3], [4, 5, 6]])
5
6 M, N = A.shape
7 U, s, Vh = linalg.svd(A)
8 sig = linalg.diagsvd(s, M, N)
9 print("U:%s\r\n*****\r\n" % U)
10 print("Vh:%s\r\n*****\r\n" % Vh)
11 print("s:%s\r\n*****\r\n" % s)
12 print("sig:%s\r\n" % sig)
13

```

程序中的 U 即为公式中的 U ， Vh 为公式中的 V^T ， s 为特征值组成的数组， sig 为公式中的 Σ ，运行结果如下：

```

U:[[-0.3863177 -0.92236578]
 [-0.92236578  0.3863177 ]]
*****

Vh:[[-0.42866713 -0.56630692 -0.7039467 ]
 [ 0.80596391  0.11238241 -0.58119908]
 [ 0.40824829 -0.81649658  0.40824829]]
*****

s:[ 9.508032  0.77286964]
*****

sig:[[ 9.508032  0.  0.  ]
 [ 0.  0.77286964  0.  ]

```

2.3 TensorFlow 简易教程

TensorFlow 是 Google 推出的开源深度学习框架，读者如果关心深度学习方面的内容，对这个开源框架一定非常熟悉，在这里就不做过多介绍了。下面向大家简单介绍一下 TensorFlow 在深度学习实际应用中用到的关键技术。TensorFlow 在实际应用中主要用于如下方面：引入或生成数据集、归一化数据、划分训练/验证/测试数据集、设置超参数、初始化变量和 placeholder、定义模型结构、定义代价函数、初始化训练网络、评估网络性能、微调超参数、实际部署运行。下面我们将分别描述在这些阶段所用到的技术。

2.3.1 张量定义

TensorFlow 中的计算是以张量 Tensor 为基础的,与外界交换信息时采用 `numpy.ndarray`。要想使用 TensorFlow,必须了解怎样定义 Tensor,定义张量 Tensor 的方法有很多,典型的有以下几种。

(1) 生成 10×20 的二维张量 bias,并以 0 进行初始化:

```
bias = tf.zeros([10, 20])
```

(2) 生成 10×20 的二维张量 v1,并以 1 进行初始化:

```
v1 = tf.ones([10, 20])
```

(3) 生成 10×20 的二维张量 v2,并以 22 进行初始化:

```
v2 = tf.fill([10, 20], 55)
```

(4) 用常量初始化张量 v3:

```
v3 = tf.constant([1, 2, 3])
v3 = tf.constant([1, 2, 3], [4, 5, 6])
```

(5) 用随机数初始化张量:

```
w1 = tf.random_uniform(shape=[10, 20],minval=0, maxval=1)
w2 = tf.random_normal(shape=[10, 20],mean=0.0, stddev=0.1)
```

在深度学习中,连接权值通常要初始化为小的随机数,在第一个例子中,我们用均匀分布在 0~1 的随机数初始化 w1;在第二个例子中,我们使用均值为 0.0、标准差为 0.1 的正态分布的随机数来初始化 w2。

但是在某些情况下,上面的正态分布随机数还是有可能取偏离均值很远的极端值,这无疑是我们所不希望看到的,我们可以采用下面的方法,将随机数控制在均值附近正负标准差的范围内。

```
w3 = tf.truncated_normal(shape=[10, 20],mean=0.0, stddev=1.0)
```

(6) 随机洗牌。

在很多情况下,我们需要随机安排输入样本顺序,可以使用如下方法:

```
shuffled_input = tf.random_shuffle(input_tensor)
```

(7) 样本裁剪。

在某些情况下,例如输入图像尺寸过大时,就需要对输入图像进行裁剪,可以采用如下方法:

```
cropped_image = tf.random_crop(image_in, [height, width, 3])
```

2.3.2 变量和 placeholder

初学 TensorFlow 时,大家可能对这两个类型区分不清,其实如果是网络参数,需要网

络学习算法进行修改的，都采用变量来表示。而需要输入给网络，不需要学习算法修改的内容，都定义为 placeholder。例如，网络的连接权值和偏置值 bias 都定义为变量，网络的输入信号和 Dropout 的失活率等一般定义为 placeholder。

我们先来看变量的使用方法，代码如下：

```
1 import numpy as np
2 import tensorflow as tf
3
4 def startup():
5     print('hello world')
6     v1 = tf.Variable(tf.zeros([2, 3]))
7     print('v1:{0}'.format(v1))
8     with tf.Session() as sess:
9         sess.run(tf.global_variables_initializer())
10        rst = sess.run(v1)
11        print(rst)
12
13 if '__main__' == __name__:
14     startup()
```

运行程序，输出结果为：

```
hello world
v1:<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32_ref>
[[ 0.  0.  0.]
 [ 0.  0.  0.]]
```

第 6 行：定义一个由 0 组成的 2×3 的二维矩阵变量 v1。

第 7 行：如果此时打印 v1 的内容，会得到输出结果中第 2 行的内容，看不到 v1 中的具体内容。

第 8 行：启动 TensorFlow 会话。

第 9 行：初始化变量。

第 10 行：对 v1 进行求值，在 TensorFlow 中想要显示变量或 Tensor 的值，都必须通过 TensorFlow 会话的 run 方法进行求值。

第 11 行：打印 v1 的内容，只有此时才能正常打印出变量的内容。

下面我们来看 placeholder 的使用方法，代码如下：

```
1 import numpy as np
2 import tensorflow as tf
3
4 def startup():
5     print('hello world')
6     x = tf.placeholder(shape=[2, 3], dtype=tf.float32)
7     y = 2*x
8     x_in = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
9     with tf.Session() as sess:
10        sess.run(tf.global_variables_initializer())
11        rst = sess.run(y, feed_dict={x: x_in})
12        print(rst)
13
14 if '__main__' == __name__:
15     startup()
```

第 6 行：定义了一个二维 2×3 的 placeholder。

第 7 行：定义了一个张量 y，其值是 x 的 2 倍。

第 8 行：用数值初始化一个 2 行 3 列的 numpy 数组。

第 9 行：启动 TensorFlow 会话。

第 10 行：初始化变量。

第 11 行：命令 TensorFlow 会话计算并返回张量 y 的值，计算过程中所需要的 x 的值由 x_in 来给出。

第 12 行：打印返回的张量 y 的值。

上面的程序中展示出的原理，就是我们在深度学习中计算网络信号前向传输最后得到输出层输出的方法。运行上面的程序的输出结果为：

```
[[ 2.  4.  6.]
 [ 8. 10. 12.]]
```

2.3.3 神经元激活函数

神经元常用的激活函数主要有：Sigmoid、ReLU、Tanh，在 TensorFlow 中使用这些函数将非常简单，代码如下：

```
1 import numpy as np
2 import tensorflow as tf
3
4 def startup():
5     print('hello world')
6     z = tf.placeholder(shape=[3], dtype=tf.float32)
7     a_sigmoid = tf.nn.sigmoid(z)
8     a_relu = tf.nn.relu(z)
9     a_tanh = tf.nn.tanh(z)
10    z_in = np.array([1.0, -2.0, 3.0])
11    with tf.Session() as sess:
12        sess.run(tf.global_variables_initializer())
13        rst_sigmoid = sess.run(a_sigmoid, feed_dict={z: z_in})
14        rst_relu = sess.run(a_relu, feed_dict={z: z_in})
15        rst_tanh = sess.run(a_tanh, feed_dict={z: z_in})
16        print('sigmoid:{0}; \r\nrelu:{1}; \r\ntanh:{2}'.format(rst_sigmoid,
17            rst_relu, rst_tanh))
18 if '__main__' == __name__:
19     startup()
```

第 6 行：定义神经元输入信号的 placeholder。

第 7 行：定义 Sigmoid 神经元输出 a_sigmoid。

第 8 行：定义 ReLU 神经元输出 a_relu。

第 9 行：定义 tanh 神经元输出 a_tanh。

第 10 行：定义网络的输入信号的具体值。

第 11 行：启动 TensorFlow 会话。

第 12 行：初始化变量。

第 13 行：求出 Sigmoid 神经元输出结果。

第 14 行：求出 ReLU 神经元输出结果。

第 15 行：求出 tanh 神经元输出结果。

第 16 行：打印 Sigmoid、ReLU、Tanh 神经元的输出结果。

运行程序后的结果如下：

```
sigmoid:[ 0.7310586  0.11920292  0.95257413];
relu:[ 1.  0.  3.];
tanh:[ 0.76159418 -0.96402758  0.99505472]
```

2.3.4 线性代数运算

在深度学习中，线性代数运算使用得非常广泛，无论是多层感知器信号前向传播过程，还是卷积神经网络将卷积操作转换为矩阵操作。不仅如此，通过对矩阵特征值分解等手段，我们还可以深入研究代价函数性质，从而有助于我们找到更接近全局最小值的解。在这一部分中，我们将向大家简单演示一下，在 TensorFlow 中怎样进行线性代数操作。

我们首先来看矩阵的运算，借助于 `numpy`，矩阵实际上就是二维数组，矩阵运算与 `numpy` 中的矩阵运算非常相似，只不过我们需要在 TensorFlow 的会话中完成计算，代码如下：

```
1 import numpy as np
2 import tensorflow as tf
3
4 def main():
5     A = tf.fill([2, 3], 3.0)
6     B = tf.fill([2, 3], 1.0)
7     C = tf.fill([3, 2], 5.0)
8     I = tf.diag([1.0, 1.0, 1.0])
9     with tf.Session() as sess:
10         rst_A = sess.run(A)
11         rst_A_p_B = sess.run(A+B)
12         print('矩阵加法: \r\n{0}'.format(rst_A_p_B))
13         rst_A_s_B = sess.run(A-B)
14         print('矩阵减法: \r\n{0}'.format(rst_A_s_B))
15         rst_A_m_C = sess.run(tf.matmul(A, C))
16         print('矩阵乘法: \r\n{0}'.format(rst_A_m_C))
17         rst_I = sess.run(I)
18         print('单位矩阵: \r\n{0}'.format(rst_I))
19         rst_A_m_I = sess.run(tf.matmul(A, I))
20         print('矩阵乘以单位矩阵: \r\n{0}'.format(rst_A_m_I))
21
22 if '__main__' == __name__:
23     main()
```

第 5 行：定义 A 为 2 行 3 列矩阵，并且元素值均为 3.0。

第 6 行：定义 B 为 2 行 3 列矩阵，并且元素值均为 1.0。

第 7 行：定义 C 为 3 行 2 列矩阵，并且元素值均为 5.0。

第8行：定义 I 为 3 行 3 列单位矩阵（主对角线上元素为 1，其他的元素为 0），这里我们使用了 TensorFlow 的 `diag` 函数，这个函数是将参数中的数组作为对角阵对角线上的元素，其他元素均为 0。

第9行：启动 TensorFlow 会话。

第10行：求出矩阵 A 的值 `rst_A`，此时才可以通过打印 `rst_A` 打印出矩阵 A 的内容，否则 A 为一个张量，没有绑定具体的值。

第11行：计算矩阵 A 加上矩阵 B 的值。

第12行：打印矩阵 A 与矩阵 B 之和。

第13行：计算矩阵 A 减去矩阵 B 的值。

第14行：打印矩阵 A 与矩阵 B 之差。

第15行：计算矩阵 A 与矩阵 B 的乘积。

第16行：打印矩阵 A 与矩阵 B 的乘积。

第17行：求出单位矩阵的值。

第18行：打印单位矩阵。

第19行：求出矩阵 A 与单位矩阵的乘积。

第20行：打印矩阵 A 与单位矩阵的乘积。

下面我们来看矩阵的特征值分解，正如整数因数分解可以让我们了解整数的性质一样，通过矩阵的特征值分解，我们同样可以了解很多矩阵的相关特性。在深度学习领域，我们经常会研究代价函数的一阶导数组成的 **Jacs** 矩阵及二阶导数组成的 **Hessian** 矩阵，得到代价函数的特性，从而更容易逼近全局最优解。

对于一个方阵 A （特征值分解仅对方阵有定义），我们定义方阵 A 的特征向量 \boldsymbol{v} 具有如下性质：

$$A\boldsymbol{v} = \lambda\boldsymbol{v}$$

式中， \boldsymbol{v} 为特征向量， λ 为该特征向量对应的特征值，代码如下：

```
1 import numpy as np
2 import tensorflow as tf
3
4 def main():
5     A = tf.constant(value=[[1, -3.0, 3.0], [3, -5, 3],
6                             [6, -5, 4]], dtype=tf.float32)
7     e, v = tf.self_adjoint_eig(A)
8     with tf.Session() as sess:
9         rst_A = sess.run(A)
10        rst_e, rst_v = sess.run([e, v])
11        print('A:\r\n{0}'.format(rst_A))
12        print('特征值: {0}'.format(rst_e))
13        print('特征向量: \r\n{0}'.format(rst_v))
14
15
16 if '__main__' == __name__:
17     main()
```

第5行：用常数定义一个矩阵。

第 7 行：利用 TensorFlow 函数 `self_adjoint_eig` 求出矩阵 `A` 的特征值 `e` 和特征向量 `v`。

第 8 行：启动 TensorFlow 会话。

第 9 行：求出矩阵 `A` 的内容。

第 10 行：求出矩阵 `A` 的特征值和特征向量。

第 11 行：打印矩阵 `A` 的内容。

第 12 行：打印特征值。

第 13 行：打印特征向量。

2.3.5 操作数据集

深度学习是通过对数据的学习，发现其中的规律，从而完成学习的过程。因此对数据集的操作是深度学习的基础和前提条件。在本书中，我们用到的数据集主要是 MNIST 手写数字识别数据集，这个数据集在深度学习领域非常通用，几乎成为每种算法证明自己效率的途径。在 TensorFlow 中，对这个数据集提供了完整的内置支持，因此不必编写代码，就可以直接使用这一数据集。下面我们来看看在 TensorFlow 中怎样读取 MNIST 手写数字识别数据集，代码如下：

```
1 import sys
2 import numpy as np
3 import argparse
4 import matplotlib.pyplot as plt
5 import matplotlib.image as mpimg
6 from skimage import io
7 import TensorFlow as tf
8 from TensorFlow.examples.tutorials.mnist import input_data
9
10 class Mnister:
11     def __init__(self):
12         pass
13
14     def learn_mnist(self):
15         mnist = input_data.read_data_sets('datasets',
16                                           one_hot=True)
17         X_train = mnist.train.images
18         y_train = mnist.train.labels
19         X_validation = mnist.validation.images
20         y_validation = mnist.validation.labels
21         X_test = mnist.test.images
22         y_test = mnist.test.labels
23         print('X_train: {0} y_train:{1}'.format(
24               X_train.shape, y_train.shape))
25         print('X_validation: {0} y_validation:{1}'.format(
26               X_validation.shape, y_validation.shape))
27         print('X_test: {0} y_test:{1}'.format(
28               X_test.shape, y_test.shape))
29         image_raw = (X_train[1] * 255).astype(int)
30         image = image_raw.reshape(28, 28)
```

```

31     label = y_train[1]
32     idx = 0
33     for item in label:
34         if 1 == item:
35             break
36         idx += 1
37     plt.title('digit:{0}'.format(idx))
38     plt.imshow(image, cmap='gray')
39     plt.show()
40
41 def main(_):
42     mnister = Mnister()
43     mnister.learn_mnist()
44
45 if '__main__' == __name__:
46     parser = argparse.ArgumentParser()
47     parser.add_argument('--data_dir', type=str, default='datasets',
48                         help='Directory for storing input data')
49     FLAGS, unparsed = parser.parse_known_args()
50     tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)

```

第 8 行：引入 TensorFlow 中 MNIST 手写数字识别库。

第 15、16 行：调用 TensorFlow 的 `input_data` 的 `read_data_sets` 方法，第一个参数为数据集存放路径，第二个参数是标签集的格式。在原始 MNIST 数据集中，我们知道每个样本是 28×28 的黑白图片，对应的是 $0 \sim 9$ 的数字标签，所以其格式为：[...784 (28×28) 像素点的值...][3]。其中，第一项为 784 (28×28) 个 $0 \sim 1$ 的浮点数，0 代表黑色，1 代表白色；第二项的“3”代表这个样本是数字 3。为了后续处理方便，我们将标签[3]改为 one-hot 形式，因为标签代表 $0 \sim 9$ 的数字，所以标签集为 10 维向量，每维上取值为 0 代表不是这个对应位置的数字，取值为 1 代表是这个对应位置的数字。其中，只有一维可以取 1，因此称之为 one-hot，还以上面的例子为例，标签集的格式就变为：[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]，因为第四位为 1，所以代表这个样本是数字 3。

第 17 行：取出训练样本集输入信号集 `X_train`，其为设计矩阵形式，每一行代表一个样本，行数为训练样本集中的样本数量。在这个例子中，就 $X_{\text{train}} \in \mathbb{R}^{55000 \times 784}$ ，其中训练样本集中有 55000 个样本，每个样本是 784 (28×28) 维的图片。

第 18 行：取出训练样本集标签集 `y_train`，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为训练样本集中的样本数量。在这个例子中，就 $y_{\text{train}} \in \mathbb{R}^{55000 \times 10}$ ，其中训练样本集中有 55000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 $0 \sim 9$ 的数字。

第 19 行：取出验证样本集输入信号集 `X_validation`，其为设计矩阵形式，每一行代表一个样本，行数为验证样本集中的样本数量。在这个例子中，就 $X_{\text{validation}} \in \mathbb{R}^{5000 \times 784}$ ，其中验证样本集中有 5000 个样本，每个样本是 784 (28×28) 维的图片。根据前面我们的讨论可以知道，在训练过程中，为了防止模型出现过拟合，模型的泛化能力降低（模型在训练样本集达到非常高的精度，但是在未见过的测试样本集或实际应用中，精度反而不高），通常会采用 **Early Stopping** 策略，就是在逻辑回归模型训练过程中，只用训练样本集对模型进行训练，每隔一定的时间间隔，计算模型在未见过的验证样本集上识别的精度，并记录

迄今为止在验证样本集上取得的最高精度。我们会发现，在训练初期，验证样本集上的识别精度会稳步提高，但是到了一定阶段之后，验证样本集上的识别精度就不会再明显提高了，甚至开始逐渐下降，这就说明模型出现了过拟合，这时就可以停止模型训练，将在验证样本集上取得最佳识别精度的模型参数作为模型最终的参数。综上所述，验证样本集主要用于防止模型出现过拟合，为 Early Stopping 算法提供终止依据。

第 20 行：取出验证样本集标签集 $y_validation$ ，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为训练样本集中的样本数量。在这个例子中，就 $y_validation \in \mathbb{R}^{5000 \times 10}$ ，其中验证样本集中有 5000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 21 行：取出测试样本集输入信号集 X_test ，其为设计矩阵形式，每一行代表一个样本，行数为训练样本集中的样本数量。在这个例子中，就 $X_test \in \mathbb{R}^{10000 \times 784}$ ，其中训练样本集中有 10000 个样本，每个样本是 784（28×28）维的图片。测试样本集主要用于模型训练结束后对模型性能进行评估。由于模型没有见过测试样本集中的样本，可以模拟模型在实际部署之后的情况，模型在测试样本集上的识别精度，基本可以视为模型在实际应用中可以达到的精度。

第 22 行：取出测试样本集标签集 y_test ，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为测试样本集中的样本数量。在这个例子中，就 $y_test \in \mathbb{R}^{10000 \times 10}$ ，其中测试样本集中有 10000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 23~28 行：分别打印训练样本集、验证样本集、测试样本集的内容。

第 29 行：以训练样本集中第 2 个样本为例，带领大家看一下 MNIST 手写数字识别数据集的具体内容。首先读出样本输入信号，其为 784（28×28）维向量，元素为 0~1 的浮点数。由于要进行显示，所以将每个元素乘以 255，变为 0~255 的浮点数。然后再将其变为 0~255 的整数。

第 30 行：将 784 维向量 $image_raw$ 转换为 28×28 的矩阵。

第 31 行：取出该样本的正确结果标签。

第 32 行：定义索引号。

第 33 行：对 one-hot 形式标签循环第 34~36 行操作。

第 34、35 行：如果标签 one-hot 向量此元素为 1，则终止循环，此时 idx 的值就是所对应的数字。

第 36 行：如果标签 one-hot 向量此元素不为 1，则索引号加 1。

第 37 行：将样本标签的正确结果显示在图片标题中。

第 38 行：以灰度图像形式显示样本对应的图片。

第 39 行：具体显示图片。

运行上面的程序，会打印出如图 2.3 所示的图形。

由图可以看出，这个图片代表的是数字 3，本书后面章节的绝大部分示例都是通过各种机器学习算法识别出类似的数字。

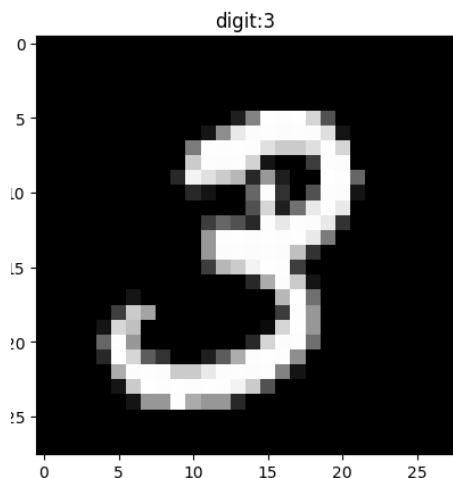


图 2.3 MNIST 手写数字识别数据集

2.4 Theano 简易教程

2.4.1 安装 Theano

我们可以通过二进制方式来安装 Theano，其过程相对简单。但是如果想对 Theano 有一个全面的了解，则需要以源码方式来安装，下面就为读者介绍如何以源码方式安装 Theano。

首先安装需要的依赖库，进入并激活前文中创建的 Python 虚拟环境。在笔者的系统中，虚拟环境安装在 `/home/osboxes/dev/wky` 目录下，则激活命令为：

```
source /home/osboxes/dev/wky/bin/activate
安装 nose: pip install nose
安装 fake8: pip install fake8
```

因为要与 Theano 开发小组的代码风格保持一致，因此需要配置文本编辑器 Vim。下载 Theano 项目的 Vim 风格文件：

```
git clone https://github.com/VundleVim/Vundle.vim.git ~/.vim/bundle/Vundle.vim
```

下面需要进行 Vim 插件安装，运行：Vim。进入 Vim 后，在命令模式下执行：PluginInstall。

完成以上步骤后，就可以开始下载 Theano 源码了。但是 Theano 源码在 GitHub 上需要使用 Git 获取，而公司项目也有可能使用 Git。在多数情况下，个人的 GitHub 账号与公司项目的有所不同，因此就需要为 GitHub 建立单独的公钥和私钥文件。

```
运行: ssh-keygen -t rsa -C your_git_hub_email@aaa.com。
```

我们需要找到密钥文件的存放目录，先进入 `~/.ssh` 目录，然后运行 `pwd` 命令，查看该目录全路径名称，例如在笔者的机器上就是 `/home/osboxes/.ssh`。

在输入文件名处输入：/home/osboxes/.ssh/id_rsa_github，其他项可以取默认设置。

登录 Theano 在 GitHub 上的主页，先 Fork 该项目，此时 GitHub 会在你的 GitHub 下创建 Theano 项目，以笔者的项目为例就是 <https://github.com/yt7589/Theano>。

进入你的 GitHub 主页，以笔者的项目为例是 <https://github.com/yt7589>，先单击“Edit Profile”按钮，然后选择“SSH and GPG keys”选项，再单击“添加”按钮，将 ~/.ssh/id_rsa_github.pub 文件中的内容复制到 Key 中。

```
运行: ssh-agent bash --login -i
运行: ssh-add /home/osboxes/.ssh/id_rsa_github
```

将 GitHub 加入已知主机列表：ssh -T git@github.com。

正式下载 Theano 源码，运行命令：

```
sudo git clone git@github.com:yt7589/Theano.git
sudo git remote add central git://github.com/Theano/Theano.git
```

因为需要与 Theano 项目进行合作，所以先与其进行关联：

```
git fetch central
git branch trunk central/master
```

接下来获取最新的源码：

```
git checkout trunk
git pull
```

全部源码下载完成之后，就要安装源码了。进入 Theano 源码目录，先运行：

```
sudo python3 setup.py develop。
```

再运行 Python，在 Python 下输入：

```
import theano
theano.test()
```

这些命令将执行 Theano 所有的测试用例，会花费很长时间，需要耐心等待。

2.4.2 Theano 入门

Theano 实际上与我们平时接触到的系统有很大的不同，在使用 Theano 的时候，并不是编写一个通常意义上的程序，而是在给 Theano 框架写脚本（虽然这种脚本是 Python 语言的）。Theano 在接收到我们所写的程序代码之后，会先将其编译成 C 代码，然后再编译成机器语言进行执行，通过这种方式，可以提高运行时的执行效率。另外，Theano 这种基于计算图式的实现方式，已经被越来越多的开源深度学习框架所采用，如 TensorFlow、MXNet 等。这种方式的优点是执行效率可以大幅提高，一方面可以转化为更加高效的 C 语言，另一方面可以将求导结果进行缓存，使复杂的求导过程复用，减少运算量。但是这种方式也有明显的缺点，就是一旦程序出现问题，我们将不能进行跟踪调试。从这一点来看，类似 Caffe 这种框架在跟踪调试方面会有一些优势。

下面将向读者演示怎样通过 Theano 框架求两个数的和。通过这样一个 Hello World 级

应用，可以大致了解 Theano 框架程序开发的基本概念和方法。

输入 chp02/b01.py 文件中的以下代码：

```
1 import numpy as np
2 import theano.tensor as T
3 from theano import function
4
5 x = T.dscalar('x')
6 y = T.dscalar('y')
7 z = x + y
8 f = function([x, y], z)
9 print(f(3, 5))
10 print(np.allclose(f(3, 6), 9))
11
```

在上面的代码中，首先引入了 NumPy 库，然后引入 tensor 模块和 function 模块。

第 5 行和第 6 行：分别定义了 x 和 y 两个变量，在 Theano 中，也可以称 x 和 y 为符号，在本书中，变量和符号这两个词的含义相同。我们可以注意到，x 和 y 的类型都是标量，即一个简单的数值，但是并没有为它们赋初值。

第 7 行：定义变量 z，其值是 x 与 y 之和。

第 8 行：定义了一个指向函数的变量 f，其定义了一个函数，该函数的输入参数是 x 和 y，输出值是 z。所以，该函数的功能是求 x 与 y 之和，并返回该值。

当程序执行到第 8 行时，Theano 就会将之前的代码先编译成 C 代码，最后转化为机器码。所以在执行 Theano 程序时，总会感觉启动时有些慢，这是因为 Theano 框架正在执行代码编译过程。

第 9 行：调用 f 变量，3 和 5 作为实参传入，系统会自动计算结果，并打印在界面中。

程序中定义的 x、y 和 z，其实都是 numpy.ndarray，但是其维度为 0，所以变成了标量值。

第 10 行：用 np.allclose 函数比较 f 计算出的 0 维数组与给定的 0 维数组是否相等，并打印出结果。

2.4.3 Theano 矩阵相加

在 Theano 中，虽然可以用 NumPy、SciPy 库很方便和高效地进行矩阵加法等运算，但是通常不直接进行计算，而是采用 Theano 框架来计算。与两个标量的加法运算类似，在做矩阵加法运算时，也是先定义两个矩阵变量 x 和 y，然后定义变量 z 为 x 和 y 之和，最后定义函数的输入参数为 x 和 y，输出值为 z，并调用定义的函数进行实际的矩阵加法运算：

```
1 import numpy as np
2 import theano.tensor as T
3 from theano import function
4
5 x = T.dmatrix('x')
6 y = T.dmatrix('y')
7 z = x + y
8 f = function([x, y], z)
9
10 a = np.array([[1, 2], [3, 4]])
```

```

11 b = np.array([[10, 20], [30, 40]])
12 print(f(a, b))
13

```

需要注意的是，`dmatrix` 和 `np.array` 实际上是一样的，所以在程序中定义 `x` 和 `y` 时，使用的是 `dmatrix`；而在实际调用时，是使用 `np.array` 来进行初始化的。

2.4.4 变量和共享变量

Theano 里面的函数是一个特别重要的概念，我们会一直使用它。为了更好地使用 Theano 中的函数，我们先来介绍一下变量。

下面这个例子可以用来计算 Logit 函数的值：

```

1 import theano
2 import theano.tensor as T
3 x = T.dmatrix('x')
4 s = 1 / (1 + T.exp(-x))
5 logistic = theano.function([x], s)
6
7 result = logistic([[1, 2], [-1, -2]])
8 print(result)
9

```

第 3、4 行：分别定义变量 `x` 和 `s`。

第 5 行：定义一个函数，输入值为 `x`，输出值为 `s`。

第 7 行：以一个矩阵初始化 `x`，调用函数 `logistic`，函数以矩阵形式返回 `s` 的值。

第 8 行：打印结果。

与程序设计语言类似，Theano 中的函数也允许定义参数的默认值，但是具有默认值的参数必须放在没有默认值的参数之前。不仅如此，在苹果系统的 Objective-C 和 Swift 语言中，还可以给参数取一个名字，参数名可以与函数名一起作为函数的标识符，Theano 中也支持与之类似的做法。代码如下（`chp02/b003.py`）：

```

1 import theano
2 import theano.tensor as T
3 from theano import In
4 from theano import function as f
5
6 x, y, w = T.dscalars('x', 'y', 'w')
7 z = (x + y)*w
8 mf = f([x, In(y, value=1), In(w, value=2, name='weight')], z)
9
10 r1 = mf(33)
11 print('r1=%s' % r1)
12 r2 = mf(33, 2)
13 print('r2=%s' % r2)
14 r3 = mf(33, 0, 1)
15 print('r3=%s' % r3)
16 r4 = mf(33, weight=1)
17 print('r4=%s' % r4)
18

```

程序运行结果：

```

r1=68.0
r2=70.0
r3=33.0
r4=34.0

```

第 6 行：定义 3 个标量（数值） x 、 y 和 w 。

第 7 行：定义变量 z ，其值为 x 、 y 和 w 的相关运算结果。

第 8 行：定义一个函数，输入参数为 x 、 y 和 w ，输出结果为 z 。在这里需要注意的是参数 y ，其采用 `In` 来描述，默认值为 1，在没有默认值的变量 x 的后面。定义参数 w 的值为 2，同时其名称为 `weight`。在实际调用此函数时，可以通过 `name` 来指定 w 变量。

第 10、11 行：仅提供 1 个参数，Theano 会认为该值为 x 的值，参数 y 和 w 取默认值。

第 12、13 行：提供 2 个参数，Theano 认为是给参数 x 和 y 赋值，参数 w 取默认值。

第 14、15 行：提供 3 个参数，Theano 分别赋值给参数 x 、 y 和 w ，不使用默认值。

第 16、17 行：如果我们希望只提供参数 x 和 w 的值，而参数 y 取默认值，则可以通过参数 w 的 `name` 来赋值给参数 w 。

Theano 的函数不仅可以返回一个数值，还可以同时返回多个值，代码如下（`chp02/b004.py`）：

```
1 import theano
2 import theano.tensor as T
3 from theano import function as f
4
5 a, b = T.dmatrices('a', 'b')
6 diff = a - b
7 abs_diff = abs(diff)
8 diff_squared = diff**2
9 mr = f([a, b], [diff, abs_diff, diff_squared])
10
11 result = mr([[1, 2], [3, 4]], [[1, 1], [1, 1]])
12 print(result)
13
```

程序运行结果：

```
[array([[ 0.,  1.],
        [ 2.,  3.]]), array([[ 0.,  1.],
        [ 2.,  3.]]), array([[ 0.,  1.],
        [ 4.,  9.]])]
```

在通常条件下，变量在函数调用时被赋值，当函数退出时，值就消失了，这种做法和其他编程语言一样。但是有时候需要函数中保留状态信息，例如统计某个函数的调用次数，那么就需要在函数中有一个变量，每次进入函数时，将该变量的值加 1。这种需求在 Theano 中是用共享变量来实现的。代码如下（`chp02/b005.py`）：

```
1 import theano
2 import theano.tensor as T
3 from theano import function as f
4 from theano import shared
5
6 state = shared(0)
7 dstate = shared(0)
8 inc = T.iscalar('inc')
9 acc = f([inc], state, updates=[
10     (state, state+inc),
11     (dstate, dstate + 2*inc)
12 ])
13
14 dec = f([inc], state, updates=[(state, state-inc)])
15
16 r1 = acc(10)
17 print('r1=%s; state=%d; dstate=%d' % (r1, state.get_value(),
18     dstate.get_value()))
19 r2 = dec(5)
20 print('r2=%s; state=%d' % (r2, state.get_value()))
21 state.set_value(100)
```

```

22 r3 = acc(200)
23 print('r3=%s; state=%d' % (r3, state.get_value()))
24

```

程序运行结果：

```

r1=0; state=10; dstate=20
r2=10; state=5
r3=100; state=300

```

第 6、7 行：分别声明了两个共享变量 `state` 和 `dstate`。

第 8 行：定义一个整数变量 `inc`。

第 9 行～第 11 行：先定义一个增加函数，输入参数为变量 `inc`，输出值为共享变量 `state` 的当前值，之后将共享变量 `state` 的值增加 `inc`，并将共享变量 `dstate` 的值增加 $2 \times inc$ 。

第 14 行：定义减少函数，输入参数为变量 `inc`，输出值为共享变量 `state` 的当前值，并将共享变量 `state` 的值减少 `inc`。从这里可以看出，共享变量 `state` 可以在增加函数 `acc` 和减少函数 `dec` 之间共享使用。

第 16～18 行：调用增加函数，并打印结果。

第 19、20 行：调用减少函数，并打印结果。

第 21～23 行：先更新共享变量 `state` 的值，然后调用增加函数，并打印相关结果。

如果表达式是用共享变量定义的，但是在表达式求值的时候却不想使用共享变量的值，而是希望临时指定一个值，可以使用 `givens` 关键字。代码如下（`chp02/b006.py`）：

```

1 import theano
2 import theano.tensor as T
3 from theano import function as f
4 from theano import shared
5
6 state = shared(3)
7 delta = T.iscalar('delta')
8 exp = state * 2 + delta
9 temp_val = T.scalar(dtype=state.dtype)
10 skip_func = f([delta, temp_val], exp, givens=[(state, temp_val)])
11
12 result = skip_func(1, 5)
13 print('result=%s; state=%d' % (result, state.get_value()))
14

```

程序运行结果：

```
result=11; state=3
```

第 6 行：定义共享变量 `state`。

第 7 行：定义普通变量 `delta`。

第 8 行：定义用到共享变量的表达式，并用变量 `exp` 表示。

第 9 行：定义一个临时变量 `temp_val`，在函数中将用这个变量替换共享变量 `state`。

第 10 行：定义 `skip_func` 函数，输入参数为变量 `delta` 和 `temp_val`，输出值为变量 `exp` 定义的表达式的值，但是该表达式是通过共享变量 `state` 计算得到的，我们在这里希望使用 `temp_val` 的值代替共享变量 `state` 的值来进行计算。为了达到这个目的，通过使用 `givens` 关键字来实现。通过 `givens` 告诉 Theano，我们希望用 `temp_val` 的值取代 `state` 的当前值来计算函数的返回值。

第 13 行：打印出了我们想要的结果。如果按照共享变量 `state` 的当前值来进行计算，则

结果应该是 7；而如果按照临时变量 `temp_val` 的值来计算，则结果应该是 11。打印出来的结果为 11，这说明 `givens` 达到了目的。

我们知道，`Theano` 中的每个函数都是需要编译的，因此会耗费一定的时间，如果两个函数基本相同，只是参数维度等有所不同，则可以使用函数复制的概念，因为复制的函数只需编译一次，可以提高效率。代码如下（`chp02/b007.py`）：

```
1 import theano
2 import theano.tensor as T
3 from theano import function as f
4 from theano import shared
5
6 state = shared(0)
7 inc = T.iscalar('inc')
8 acc = f([inc], state, updates=[(state, state+inc)])
9 new_state = shared(0)
10 new_acc = acc.copy(swap={state: new_state})
11 null_acc = acc.copy(delete_updates=True)
12
13 r1 = acc(10)
14 print('r1=%s; state=%d' % (r1, state.get_value()))
15 r2 = new_acc(100)
16 print('r2=%s; new_state=%d; state=%d' % (r2, \
17     new_state.get_value(), state.get_value()))
18 r3 = null_acc(200)
19 print('r3=%s; state=%d' % (r3, state.get_value()))
20
```

程序运行结果：

```
r1=0; state=10
r2=[array(0)]; new_state=100; state=10
r3=[array(10)]; state=10
```

第 6 行：定义共享变量 `state`。

第 7 行：定义变量 `inc`。

第 8 行：定义增加函数 `acc`，输入参数为变量 `inc`，输出值为共享变量 `state` 的当前值，返回该值后，将共享变量 `state` 的值加上 `inc`。

第 9 行：定义新的共享变量 `new_state`。

第 10 行：定义 `new_acc` 函数，该函数复制自 `acc` 函数，只是将函数中所用的共享变量从 `state` 变为 `new_state`，其余内容均不变。

第 11 行：定义 `null_acc` 函数，该函数也复制自 `acc` 函数，不过这次不需要更新共享变量 `state` 的值，其实就是内容都不变。

第 13、14 行：调用 `acc` 函数并打印结果，我们看到其返回了 `state` 的当前值，并且之后 `state` 的值也进行了正确的更新。

第 15、16 行：调用复制的 `new_acc` 函数，这时返回的是共享变量 `new_state` 的当前值，而且更新的是 `new_state` 的值，而共享变量 `state` 的值不改变。

第 18、19 行：调用复制的 `null_acc` 函数，这时只返回共享变量 `state` 的值，但是不更新共享变量 `state` 的值。

程序运行结果验证了程序逻辑。

2.4.5 随机数的使用

在深度学习中，经常会用到随机数，例如在权值矩阵初始化时。下面就来研究一下，在 Theano 中怎样产生和使用随机数。

由于在 Theano 中先采用符号定义我们需要的逻辑，然后通过编译得到可执行的函数，所以随机数的使用方法通常与在编程语言中的使用方法不同。在 Theano 中使用随机数，与前文介绍的使用共享变量类似。

在 Theano 的函数中使用随机数的代码如下（chp02/b008.py）：

```
1 import theano
2 import theano.tensor as T
3 from theano import function as f
4 from theano.tensor.shared_randomstreams import RandomStreams
5
6 srng = RandomStreams(seed=234)
7 rv_u = srng.uniform((2, 2))
8 rv_n = srng.normal((2, 2))
9 f1 = f([], rv_u)
10 g1 = f([], rv_n, no_default_updates=True)
11 nz = f([], rv_u + rv_n - 2*rv_u)
12
13 print('random1:\r\n%s' % f1())
14 print('random2:\r\n%s' % f1())
15 print('nearly zeor:\r\n%s' % nz())
16
```

程序运行结果：

```
[[ 0.12672381  0.97091597]
 [ 0.13989098  0.88754825]]
random2:
[[ 0.31971415  0.47584377]
 [ 0.24129163  0.42046081]]
nearly zeor:
[[ 0.  0.]
 [ 0.  0.]]
```

第 4 行：在 Theano 中使用随机数时，需要引入 shared_randomstreams。

第 6 行：定义 srng 是一个随机数生成引擎。

第 7 行：产生一个均匀分布的由随机数组成的 2×2 矩阵。

第 8 行：产生一个正态分布的由随机数组成的 2×2 矩阵。

第 9 行：定义函数 f1，生成均匀分布的随机数，并且在每次调用时所取的随机数不同。

第 10 行：定义函数 g1，由于禁止随机数引擎更新，所以取得的随机数相同，除非重新 seed 随机数引擎。

第 11 行：定义一个近似为 0 的矩阵，用于验证随机数生成的精度。

2.4.6 Theano 求导

在深度学习中，最常用的算法就是随机梯度下降算法，而这个算法的核心就是求偏导数。Theano 中提供了函数，可以非常方便地求出指定函数的导数。

下面先来复习一下导数的概念，例如对函数 $y = x^2$ 求导，则结果为 $\frac{dy}{dx} = 2x$ 。

如果求该函数在 $x = 2$ 上的导数，则：

$$\frac{dy}{dx}|_{x=2} = 2x|_{x=2} = 4$$

求导运算在 Theano 中可以很方便地实现，代码如下（chp02/b009.py）：

```
1 import numpy as np
2 import theano
3 import theano.tensor as T
4 from theano import function as f
5
6 x = T.dscalar('x')
7 y = x**2
8 gy = T.grad(y, x)
9 dyx = f([x], gy)
10
11 print(dyx(2))
12
```

程序运行结果：

```
4.0
```

在深度学习中，我们常用的函数是 Sigmoid 函数，其定义如下：

$$y = \frac{1}{1 + e^{-x}}$$

其导数为：

$$\frac{dy}{dx} = y(1 - y)$$

该函数的导数不是由 x 来表示的，而是由 y 来表示的。同样可以用上面的方法求出 Sigmoid 函数的导数。代码如下（chp02/b010.py）：

```
1 import numpy as np
2 import theano
3 import theano.tensor as T
4 from theano import function as f
5
6 x = T.dscalar('x')
7 y = 1 / (1 + T.exp(-x))
8 gy = T.grad(y, x)
9 dyx = f([x], gy)
10
11 print(dyx(2.0))
12
```

程序运行结果：

```
0.10499358540350649
```

由上面的代码可以看出，在 Theano 中求导是一件很容易的事情，即使像 Sigmoid 这种复杂的函数，也可以很简单地对其进行求导计算。

2.5 线性回归

有了 NumPy、SciPy 和 Theano 框架的准备知识，下面就可以进行深度学习算法的开发了。在这里，我们先从最简单的线性回归开始，向读者展示怎样使用 Theano 实现线性回归算法。

本节将从算法的理论知识开始讲起，会有较大篇幅的数学演算，然后才是利用 Theano 算法的实现部分。之所以要用较大的篇幅来讲解数学推导过程，是因为线性回归算法可以说是机器学习中最简单的算法，我们必须完整地理解这一算法。如果不能理解这一算法，后续更复杂的深度学习算法就更无从下手了。如果不理解算法背后的理论，只满足于应用开源框架运行一个个 Demo，是不能称机器学习已经入门了的。

本节的应用实例，我们选择了一个初创公司估值模型。为了更形象地理解这一问题，假设初创公司估值只依赖于一个变量，即专家评分。在训练样本中，有一系列专家评分与估值的对应关系，算法的任务就是给出一个新公司的专家评分，通过算法给出正确的估值。训练数据如表 2.1 所示。

表 2.1 训练数据

专家评分（100分制）	估值（万元）
80	1610
85	1710
60	1190
70	1390
75	1490
78	1570
.....

我们的任务是通过这些训练数据和算法学习找到其中的规律，当拿出一家新公司的专家评分时，例如专家评分为 82，我们就能预测出该公司的准确估值。

这个问题是一个超简单的问题，实际影响公司估值的因素非常多，例如创始人背景、所处行业、业务发展情况等，在这里要将所有的情况综合考虑，最终给出一个专家评分，并给出估值。所以问题的难度不在于最后估值的预测，而在于怎样得到这个专家评分。而这正是深度学习要解决的问题，类似的情况我们可以用将要讲到的堆叠去噪自动编码机和深度信念网络来解决。

2.5.1 问题描述

- 下面先定义以下符号：
- 假设训练样本共有 m 个。

- ❑ 用 \mathbf{x} 表示输入向量。
- ❑ 用 y 代表输出变量，也就是目标变量。
- ❑ 训练样本可以表示为 (\mathbf{x}, y) 。
- ❑ 假设有 m 个训练样本，则第 i 个训练样本可以表示为： $(\mathbf{x}^{(i)}, y^{(i)})$ 。

线性回归学习过程可以用图 2.4 表示。

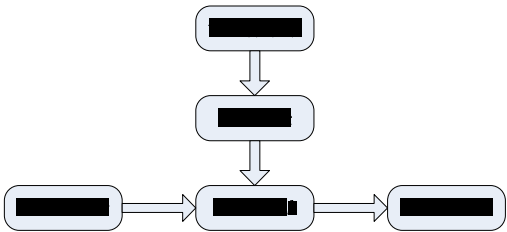


图 2.4 线性回归框架图

用训练样本集训练学习算法，从而找到最佳的假设函数 h ，这就是训练过程。当训练过程完成之后，我们拿到一个新公司的专家评分，将其作为假设函数 h 的输入，就可以求出该新公司的估值。这就是线性回归的整个过程，即包括学习过程和实际运行两个阶段。

读者可能会有疑问，我们想要学习的是深度学习，为什么要讲线性回归算法？这和深度学习甚至神经网络好像没有任何关系呀！

其实这是一种典型的误解，笔者之前也有过这种想法，结果在老师讲解这部分知识的时候没好好听讲，总想着快点结束，好学神经网络的内容。结果这部分没学好，理解其他神经网络算法时就很困难。因为线性回归算法是大多数机器学习算法的基础，只有打好基础才有可能理解其他机器学习算法。

其实，虽然线性回归算法不是深度学习算法，但是它也是一种神经网络，其可以表示成如图 2.5 所示的简单神经网络。

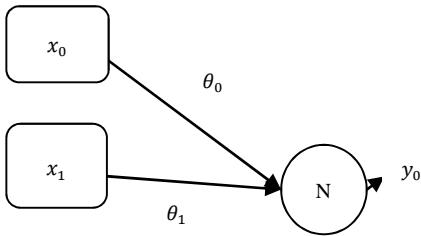


图 2.5 由线性回归算法表示的简单神经网络

图 2.5 中的圆形部分代表一个神经元，其有两个输入，分别为 x_0 和 x_1 ；还有两个连接权值，分别为 θ_0 和 θ_1 ，其网络输入为：

$$a = \theta_0 x_0 + \theta_1 x_1 = \sum_{i=0}^n \theta_i x_i$$

式中， $n=1$ ，这个神经元的激活函数可以定义为：

$$y=a$$

通过上面的假设之后，线性回归过程就变成一个典型的神经网络模型了。本节的重点不是讲述线性回归算法的神经网络表现形式，只是说明线性回归算法也是神经网络最简单的一种表现形式。

2.5.2 线性模型

下面回到正题，因为我们研究的是线性回归问题，所以假设函数 h 就是一个线性函数。因为前文所述的公司估值问题只有一个输入变量 x ，所以可以把假设函数 h 表示为如下形式：

$$h(x) = \theta_0 + \theta_1 x_1$$

但是为了后续便于用矩阵来表示，我们引入一个假的输入 $x_0 = 1$ ，这时就可以得到一个通用表达式：

$$h(x) = \theta_0 x_0 + \theta_1 x_1 = \sum_{i=0}^n \theta_i x_i = \theta^T x$$

注意：式中粗体小写字母 x 代表的是向量，其 $n=1$ ，向量维度为 2。
在前面的模型中我们只考虑了专家评分一项，假设现在还需要考虑的因素是融资轮次，那么训练数据就如表 2.2 所示。

表 2.2 训练数据

专 家 评 分	轮 次	估 值
85	1	800
.....

其中，轮次1——种子轮；轮次2——天使轮；轮次3——A轮；轮次4——B轮；轮次5——C轮；轮次6——IPO。

那么模型就可以表示为：

$$h(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 = \sum_{i=0}^n \theta_i x_i = \theta^T x$$

式中， $n=2$ ，是特征的数量。

其实引入多个特征和一个特征的处理方式基本相同，所以为了讨论问题方便起见，就只讨论一个特征的情况，读者可以很方便地将算法推广到有多个特征的情况。

对于第 i 个训练样本，定义误差为：

$$\frac{1}{2} (h_{\theta}(x^{(i)}) - y^{(i)})^2$$

式中的 $1/2$ 是为了计算方便引入的常量，会在后续推导中消去。

对于所有 m 个训练样本，我们可以得到全部的误差为：

$$\frac{1}{2} \sum_{i=1}^m (h_{\theta}(\mathbf{x}^{(i)}) - y^{(i)})^2$$

使用线性回归算法就是调整参数 θ ，使上式的值最小，将其定义为函数 J ：

$$J(\theta) = \frac{1}{2} \sum_{i=1}^m (h_{\theta}(\mathbf{x}^{(i)}) - y^{(i)})^2$$

使函数 J 最小有两种方法：迭代法和解析法。由于本节主要介绍的是机器学习算法，所以重点讲解迭代法，解析法将在后面进行简单介绍。

2.5.3 线性回归学习算法

先初始化参数向量 θ ，例如将其初始化为 0 或非常接近 0 的随机数，然后计算出误差函数 J 。如果误差值大于理想的阈值，就调整参数向量，使误差向减小的方向发展，循环往复下去，直到得到一个最优的参数向量，使误差最小。

以上就是线性回归算法的基本过程，下面来看看具体的计算过程。

对于每个训练样本，我们可以根据误差调整第 i 个参数，其为：

$$\theta_i = \theta_i - \alpha \frac{\partial}{\partial \theta_i} J(\theta) \quad (3)$$

式中， α 为学习率，为 0 至 1 之间的数，是事先人为指定的。

上式主要是对函数 J 求偏导，下面来具体推导一下这一过程：

$$\frac{\partial}{\partial \theta_i} J(\theta) = \frac{\partial}{\partial \theta_i} \frac{1}{2} (h_{\theta}(\mathbf{x}^{(i)}) - y^{(i)})^2$$

根据高等数学中关于函数求导的法则，对式 (3) 可以采用链式求导法则，即先对 $h_{\theta}(\mathbf{x}^{(i)}) - y^{(i)}$ 整体求导，然后再对其自身求导：

$$\frac{1}{2} \times 2 \times (h_{\theta}(\mathbf{x}) - y) \frac{\partial}{\partial \theta_i} (h_{\theta}(\mathbf{x}) - y)$$

将 $h_{\theta}(\mathbf{x})$ 展开可得：

$$\frac{\partial}{\partial \theta_i} J(\theta) = (h_{\theta}(\mathbf{x}) - y) \frac{\partial}{\partial \theta_i} (\theta_0 x_0 + \theta_1 x_1 + \cdots + \theta_n x_n - y)$$

因为只对参数 θ_i 求偏导，其他参数项求偏导为 0，所以可以得到：

$$\frac{\partial}{\partial \theta_i} J(\theta) = (h_{\theta}(x) - y)x_i \quad (4)$$

将式（4）代入式（3）可得：

$$\theta_i = \theta_i - \alpha(h_{\theta}(x) - y)x_i$$

以上就是线性回归算法针对单个训练样本的参数调整公式，上面的推导过程非常详细，希望读者能够理解。将来会有更复杂的算法，所以理解上面的推导过程，一方面可以为后续学习打下基础，另一方面也是一项测试。如果理解不了，那么想再深入学习就非常困难了。

以上只是针对一个训练样本的参数调整公式，如果希望先对整个训练样本集进行运算，再调整参数向量 θ ，那么将使用被称为批量学习的方法。

如果考虑全部 m 个训练样本，则参数调整公式为：

$$\theta_i = \theta_i - \alpha \sum_{j=1}^m (h_{\theta}(x^{(j)}) - y^{(j)})x_i^{(j)}$$

批量学习算法一次处理所有训练样本，根据整体误差对参数进行调整，这对于小训练样本集来说效率较高。

但是如果样本集很大，每调整一次参数都需要遍历整个训练样本集求出误差，运算量非常大。虽然线性回归问题不存在局部最小值问题，但是对于复杂的问题，批量学习算法容易陷入局部最小值，而不是全局最小值。

所以在当前主流的算法实现中，还是以单个训练样本误差决定参数调整为主，这种方法被称为在线学习，对应的学习算法被称为随机梯度下降算法。由于训练样本集的随机性，与批量学习算法相比，这种算法陷入局部最小值的可能性将大大减少。但是在线学习也有缺点，由于训练样本的随机性，不能保证每次参数值调整都是向着全局最小的方向进行的，实际情况可能是以一个“之”字形的路线向最小值点前进。为了避免这种情况，实际中大量应用迷你批量学习算法，就是将全部训练样本划分为小的批次，每个批次只有几十到几百个训练样本，根据批次的误差来调整参数值，这样既利用在线学习的随机性解决了陷入局部最小值的问题，也在一定程度上避免了因为训练样本的随机性而造成参数值调整效率不高的问题。

2.5.4 解析法

迭代法比较适合复杂的应用场景，对于简单的应用场景，例如我们现在研究的线性回归算法，解析法是更高效的解决方案。所以，在这里向读者介绍一下解析法的推导过程，也帮助读者熟悉一下深度学习中所用到的数学知识。

首先介绍向量求导符号：

$$\nabla_{\theta} J = \begin{bmatrix} \frac{\partial J}{\partial \theta_0} \\ \frac{\partial J}{\partial \theta_1} \\ \dots \\ \frac{\partial J}{\partial \theta_n} \end{bmatrix} \in \mathbf{R}^{n+1}$$

引入上面的符号之后，参数调整公式就变为：

$$\theta = \theta - \alpha \nabla_{\theta} J \in \mathbf{R}^{n+1}$$

设一个映射 f ，其定义为：

$$f: \mathbf{R}^{m \times n} \rightarrow \mathbf{R}^{m \times n}$$

该函数映射可以应用于矩阵 A ，其定义为：

$$f(A), A \in \mathbf{R}^{m \times n}$$

可以将矩阵求偏导定义为：

$$\nabla_A f(A) = \begin{bmatrix} \frac{\partial f}{\partial A_{11}} & \dots & \frac{\partial f}{\partial A_{1n}} \\ \dots & \dots & \dots \\ \frac{\partial f}{\partial A_{m1}} & \dots & \frac{\partial f}{\partial A_{mn}} \end{bmatrix}$$

对于矩阵 $A \in \mathbf{R}^{m \times n}$ ，可以定义矩阵的迹为：

$$\text{tr} A = \sum_{i=1}^n A_{ii}, A \in \mathbf{R}^{n \times n}$$

矩阵的迹具有以下性质：

- ☐ $\text{tr} AB = \text{tr} BA$
- ☐ $\text{tr} ABC = \text{tr} CAB = \text{tr} BCA$
- ☐ $\nabla_A \text{tr} AB = B^T$
- ☐ $\text{tr} A = \text{tr} A^T$
- ☐ 对于 $a \in \mathbf{R}$ ，则 $\text{tr}(a) = a$
- ☐ $\nabla_A \text{tr} ABA^T C = CAB + C^T AB^T$

这些关于矩阵迹的性质将在后续的公式推导中用到，先在这里罗列出来，其实这些性质都是可以证明的，但是因为本书重点不是讲解数学知识，因此对证明过程感兴趣的读者可以查看其他相关资料。

下面把 m 个 n 维的训练样本向量组成设计矩阵 (Design Matrix)：

$$X = \begin{bmatrix} (\mathbf{x}^{(1)})^T \\ \dots \\ (\mathbf{x}^{(m)})^T \end{bmatrix}$$

那么可以得到式：

$$\mathbf{X}\boldsymbol{\theta} = \begin{bmatrix} (\mathbf{x}^{(1)})^T \\ \dots \\ (\mathbf{x}^{(m)})^T \end{bmatrix} \boldsymbol{\theta} = \begin{bmatrix} (\mathbf{x}^{(1)})^T \boldsymbol{\theta} \\ \dots \\ (\mathbf{x}^{(m)})^T \boldsymbol{\theta} \end{bmatrix} = \begin{bmatrix} h_{\boldsymbol{\theta}}(\mathbf{x}^{(1)}) \\ \dots \\ h_{\boldsymbol{\theta}}(\mathbf{x}^{(m)}) \end{bmatrix}$$

而此时训练样本的目标输出可以表示为：

$$\mathbf{y} = \begin{bmatrix} \mathbf{y}^{(1)} \\ \dots \\ h_{\boldsymbol{\theta}}(\mathbf{x}^{(m)}) \end{bmatrix}$$

可以把线性回归过程表示为：

$$\mathbf{X}\boldsymbol{\theta} - \mathbf{y} = \begin{bmatrix} h_{\boldsymbol{\theta}}(\mathbf{x}^{(1)}) - \mathbf{y}^{(1)} \\ \dots \\ h_{\boldsymbol{\theta}}(\mathbf{x}^{(m)}) - \mathbf{y}^{(m)} \end{bmatrix}$$

因为有：

$$\mathbf{a}^T \mathbf{a} = \sum_{i=1}^n a_i^2, \quad \mathbf{a} \in \mathbf{R}^n$$

所以可以将误差函数 J 表示为矩阵和向量运算的形式：

$$J(\boldsymbol{\theta}) = \frac{1}{2} \sum_{i=1}^m (h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)})^2 = \frac{1}{2} (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})^T (\mathbf{X}\boldsymbol{\theta} - \mathbf{y}) \quad (5)$$

根据高等数学导数相关知识，要想求误差函数 J 的极值，只需对 J 求导，并令导数为 0，这样就可以求出极值点：

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = 0$$

下面进行公式的推导，因为式（5）的结果为一个数值，根据迹的性质，一个数的迹还等于一个数，因此有：

$$\nabla_{\boldsymbol{\theta}} \frac{1}{2} (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})^T (\mathbf{X}\boldsymbol{\theta} - \mathbf{y}) = \nabla_{\boldsymbol{\theta}} \frac{1}{2} \text{tr}[(\mathbf{X}\boldsymbol{\theta} - \mathbf{y})^T (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})]$$

将其展开可得到：

$$\frac{1}{2} \nabla_{\boldsymbol{\theta}} \text{tr}(\boldsymbol{\theta}^T \mathbf{X}^T - \mathbf{y}^T)(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}) = \frac{1}{2} \nabla_{\boldsymbol{\theta}} \text{tr}(\boldsymbol{\theta}^T \mathbf{X}^T \mathbf{X} \boldsymbol{\theta} - \boldsymbol{\theta}^T \mathbf{X}^T \mathbf{y} - \mathbf{y}^T \mathbf{X} \boldsymbol{\theta} + \mathbf{y}^T \mathbf{y})$$

因为最后一项与参数 $\boldsymbol{\theta}$ 无关，对 $\boldsymbol{\theta}$ 求导其将为 0，所以只需要处理前 3 项。根据迹的性质，对第一项进行处理，第二项和第三项互为转置，而矩阵和其转置的迹是相等的，综合上面的结论可以得到：

$$\frac{1}{2}(\nabla_{\theta}\text{tr}\theta\theta^{\text{T}}X^{\text{T}}X - \nabla_{\theta}\text{tr}y^{\text{T}}X\theta - \nabla_{\theta}\text{tr}y^{\text{T}}X\theta)$$

在这里设 $A=\theta$ 且 $B=I$ (单位矩阵), $C=X^{\text{T}}X$, 则根据迹的性质, 可以得到:

$$\frac{1}{2}\left((X^{\text{T}}X\theta I + X^{\text{T}}X\theta I^{\text{T}}) - \nabla_{\theta}\text{tr}2y^{\text{T}}X\theta\right) = \frac{1}{2}\left((X^{\text{T}}X\theta + X^{\text{T}}X\theta) - 2\nabla_{\theta}\text{tr}(y^{\text{T}}X\theta)\right)$$

对等式右侧第一项合并同类项, 第二项应用迹的性质, 可得:

$$\frac{1}{2}\left(2X^{\text{T}}X\theta - 2\nabla_{\theta}\text{tr}(\theta y^{\text{T}}X)\right) = 0$$

如果令 $A=\theta$, $B=y^{\text{T}}X$, 应用迹的性质, 同时对第一项求导, 可得:

$$X^{\text{T}}X\theta - X^{\text{T}}y = 0$$

可以解出参数 θ 的值:

$$\theta = (X^{\text{T}}X)^{-1}X^{\text{T}}y$$

上式就是用最小二乘法求出的参数向量的解析解, 但是要求 $X^{\text{T}}X$ 必须可逆。在一般情况下, $X^{\text{T}}X$ 是可逆的, 但是如果含有互相依赖的特征, 例如 $x_2 = 2x_1$, 这时实际上只是一个特征, 则此时可能存在 $X^{\text{T}}X$ 不可逆的情况。不过在一般情况下, 我们还是可以放心使用上式的。

2.5.5 Theano 实现

下面将讨论怎样用 Theano 实现线性回归算法。以公司估值为例, 设专家评分为 x_1 , 需要学习的估值公式为:

$$y = \theta_0 x_0 + \theta_1 x_1$$

其中, $\theta_0 = 300$ 、 $x_0 = 1$ 、 $\theta_1 = 20$, 为了让问题更加接近真实情况, 我们会在上式的计算结果上加上 0~20 的随机噪声:

$$y = \theta_0 x_0 + \theta_1 x_1 + \epsilon$$

在上式中, 专家评分为 x_1 , 采用 100 分制。我们先通过程序生成一批原始实验数据, 然后将数据向左平移 50, 再将 x 值和 y 值同时缩小 100 倍, 得到样本在 $[-0.5, 0.5]$ 。

为什么要将数据做平移和缩放呢? 直接用原始数据不行吗? 其实这就是所谓的深度学习经验了。现在计算机采用二进制表示浮点数和双精度数, 对于小数的表示是有精度限制的, 同时表示数的范围也是有限制的, 在实际中这样做会导致溢出现象。而在线性回归算法中, 一般取学习率为 0.1 左右, 如果数值太大或太小都有可能出现溢出。如果读者直接将这里的数据输入到线性回归算法中, 就很可能出现溢出, 即求不出结果。所以需要对数据进行规整, 因为专家评分范围是 $[0, 100]$, 专家评分减去 50 后, 其范围就变为 $[-50, 50]$, 再将其

除以 100，范围就变成 $[-0.5, 0.5]$ ，这时再用线性回归算法来处理就不容易出现溢出。

其实，不光在线性回归算法中需要对原始训练样本进行规整。使用其他深度学习算法时，同样需要对训练样本进行归整，只不过会利用训练样本的期望和方差来对训练样本进行规整。关于这个问题，在讲解深度学习算法相关章节时，再向读者详细介绍。

1. 生成实验数据

首先定义线性回归数据生成类 `Lr_Data_Generator`，并定义类实例属性，代码如下（b011/lr_data_generator.py 1）：

```
1 import numpy as np
2 import theano
3 import theano.tensor as T
4 from theano.tensor.shared_randomstreams import RandomStreams
5 from theano import function as f
6
7 class Lr_Data_Generator(object):
8     def __init__(self):
9         self.srng = RandomStreams(seed=234)
10        self.rv_u = self.srng.uniform((1, 1))
11        self.get_random = f([], self.rv_u)
12        self.m = 10
13        self.n = 1
14        self.v0 = 1
15        self.theta0 = 300
16        self.theta = [20]
17        self.theta0_n = 0
18        self.theta_n = [0.0]*self.n
19
```

第 7 行：定义线性回归数据生成类 `Lr_Data_Generator`。

第 8 行：定义其构造函数，通常在构造函数中定义类的实例属性。

第 9 行：定义随机数生成引擎。

第 10 行：定义基于均匀分布的随机数生成器。

第 11 行：定义随机数生成函数。

第 12 行：规定训练样本的数量 $m=10$ 。

第 13 行：定义特征数量，这里 $n=1$ ，即只有一个 $0\sim100$ 的专家评分。

第 15 行：定义前文中用到的参数 θ_0 。

第 16 行：定义前文中用到的参数向量 θ ，这里是一维向量。

第 17 行：定义规整后的参数 θ_0 的值。

第 18 行：定义规整后的参数向量 θ 的值。

接着需要定义获取专家评分的方法，代码如下（b011/lr_data_generator.py 2）：

```
51 def get_expert_score(self):
52     score = self.get_random() * 100
53     return round(score[0][0])
54
```

即先产生 $0\sim1$ 的随机数，将其乘以 100 后再四舍五入，返回整数。

在生成实验数据时，需要在生成值上添加一个随机噪声，这里设其为 $[-10,10]$ 之间的一个数 ϵ ，代码如下（b011/lr_data_generator.py 3）：

```
47 def get_kexi(self):
48     kexi_raw = self.get_random() * 20
49     return round(kexi_raw[0][0]) - 10
50
```

下面是生成训练样本的程序，训练样本由 10 个样本组成，对于原始数据是先由 `get_expert_score` 方法获取 0~100 的专家评分，再通过 `get_kexi` 方法获取随机数 ϵ ，最后根据 $y = \theta_0 + \theta_1 x_1 + \epsilon$ 计算出估值。因为需要对数据进行规整，将原始数据左移 50 并缩小 100 倍，最终得到线性回归算法所用的训练样本。代码如下：

```

20 def normalize_data(self):
21     self.theta0_n = (self.theta[0]*50 + self.theta0) / 100
22     self.theta_n[0] = self.theta[0]
23     raw = [[0.0]*self.m, [0.0]*self.m]
24     raw_sum = 0
25     X = [[0.0]*self.m, [0.0]*self.m]
26     sum = 0
27     points = [0.0]*self.m
28     raw_points = [0.0]*self.m
29     for i in range(1, self.m+1):
30         X[0][i-1] = [0.0]*self.n
31         raw[0][i-1] = [0.0]*self.n
32         sum = self.theta0_n
33         raw_sum = self.theta0
34         for j in range(self.n):
35             v1 = self.get_expert_score()
36             X[0][i-1][j] = (v1 - 50) / 100
37             raw[0][i-1][j] = v1
38             sum = sum + self.theta_n[j]*X[0][i-1][j]
39             raw_sum = raw_sum + self.theta[j]*raw[0][i-1][j]
40         v2 = self.get_kexi()
41         X[1][i-1] = sum + v2 / 100
42         raw[1][i-1] = raw_sum + v2
43         raw_points[i-1] = raw[0][i-1][0]
44         points[i-1] = X[0][i-1][0]
45     return (X, raw, points, raw_points)
46

```

第 21、22 行：因为要对原始数据进行左移 50 并缩小 100 倍，所以模型函数为 $y = \theta_0 + \theta_1 x_1$ 的参数将发生变化。下面将一步步演示推导过程，首先是向左移 50，新坐标 x_{1n} 的值为：

$$x_{1n} = x_1 - 50 \quad (6)$$

解出 x_1 得：

$$x_1 = x_{1n} + 50 \quad (7)$$

将式 (7) 带入 $y = \theta_0 x_0 + \theta_1 x_1$ ，并且令 $x_0 = 1$ ：

$$y = \theta_0 + \theta_1 (x_{1n} + 50) = (\theta_0 + 50\theta_1) + \theta_1 x_{1n} \quad (8)$$

由于要保证图形不变形，所以 X 轴和 Y 轴坐标要同时缩小 100 倍，则 X 轴坐标为：

$$x_{1f} = \frac{x_{1n}}{100} \Rightarrow x_{1n} = 100x_{1f} \quad (9)$$

Y 轴坐标为：

$$y_f = \frac{y}{100} \Rightarrow y = 100y_f \quad (10)$$

将式 (9) 和式 (10) 带入式 (8) 得：

$$100y_f = (\theta_0 + 50\theta_1) + \theta_1 100x_{1f}$$

将等式两边同时除以 100 得：

$$y_f = (\theta_0 + 50\theta_1)/100 + \theta_1 x_{1f}$$

上式就是代码第 21、22 行的计算公式，即求出平移和缩放后参数的值。对于给定的值，在 $\theta_0 = 300$ 、 $\theta_1 = 20$ 的情况下，平移缩小后 $\theta_0 = 13$ 、 $\theta_1 = 20$ 。

第 23、24 行：初始化数组，用于保存原始数据。

第 25、26 行：用于存放平移缩小后归整化的数据。

第 27 行：归整后的 X 轴坐标（用于图形绘制）。

第 28 行：原始数据下的 X 轴坐标点集（用于图形绘制）。

第 29 行：对 m 个训练样本进行循环，生成每一个训练样本。

第 30 行：对规整后的数据，根据特征数量 n 分配数组，保存输入信号值。

第 31 行：对原始数据，根据特征数量 n 分配数组，保存输入信号值。

第 32 行：为 sum 赋值为平移缩小后新的 θ_0 值，sum 为计算估值用到的辅助变量。

第 33 行：为 raw_sum 赋初值 θ_0 ，raw_sum 为计算原始数据对应的估值用到的辅助变量。

第 34~39 行：针对每个特征进行循环，首先通过调用 get_expert_score 方法获取原始特征值 v1，将其乘以对应的 θ_1 ，并累加到 raw_sum 中；再将原始特征值通过 $(v1-50)/100$ 计算得到平移缩小后的特征值，将其与新平移缩小后对应的 θ_1 相乘，并累加到 sum 中。

第 40 行：获取 $[-10,10]$ 之间的随机变量 v2。

第 41 行：将 $v2/100 + \text{sum}$ 作为最终估值。

第 42 行：将 $v2 + \text{raw_sum}$ 作为估值。

第 43 行：将原始特征值作为原始的水平坐标散点。

第 44 行：将平移缩小后的特征值作为归整后的水平坐标散点。

2. 线性回归算法实现

首先需要定义一些常用的变量（b011/lr_engine.py 1）：

```
1 import matplotlib.pyplot as plt
2 import matplotlib.animation as animation
3 from matplotlib import style
4 import theano
5 from theano import tensor as T
6 import numpy as np
7 from lr_data_generator import Lr_Data_Generator
8
9 alpha = 0.1
10 m = 10 # number of training samples
11 n = 1 # the dimension of input
12 MAX_STEPS = 1000
13 x = T.dvector('x')
14 y = T.scalar()
15
```

第 9 行：定义学习率为 0.1。

第 10 行：定义共有 10 个训练样本。

第 11 行：定义特征维度为 1。这里虽然定义特征维度为 1，但是程序都是通过向量运算来进行的，所以此处的程序均可以处理多个特征值的线性回归程序。

第 12 行：定义最大迭代次数，因为线性回归过程特征个数很少，一般几十次就可以收敛了。

第13行：定义输入特征向量。

第14行：定义最终估算值。

接着生成训练数据（b011/lr_engine.py 2）：

```
16 lr_data_generator = Lr_Data_Generator()
17 trains, trains_raw, trains_x, raw_points = lr_data_generator.normalize_data()
18
```

这里调用前文中编写的 Lr_Data_Generator 类来产生训练数据。

下面定义假想模型（b011/lr_engine.py 3）：

```
19 theta = theano.shared(np.asarray([[0.0]*n], dtype=theano.config.floatX))
20 theta0 = theano.shared(0.0)
21 h = theta0 + T.dot(theta, x)
```

第19行：定义一个向量 θ ，为对应每个特征的参数。

第20行：定义 θ_0 ，是一个偏移量。

第21行：定义假想模型 $h = \theta_0 + \theta^T x$ 。

下面定义线性回归的学习算法，代码如下（b011/lr_engine.py 4）：

```
23 J = T.mean(1/2 * T.sqr(h-y))
24 g0 = T.grad(cost=J, wrt = theta0)
25 gradient = T.grad(cost=J, wrt = theta)
26 updates = [
27     [theta, theta-gradient * alpha],
28     [theta0, theta0-g0*alpha]
29 ]
30 train = theano.function(inputs=[x, y], outputs=h,
31                          updates = updates, allow_input_downcast= True)
32
```

第23行：定义代价函数，是训练样本的均值误差平方的均值。虽然下面将使用在线学习算法，即一次只处理一个训练样本，但是如果处理所有训练样本，这段代码是通用的。

第24行：求代价函数 J 对 θ_0 的偏导数。

第25行：求代价函数 J 对参数向量 θ 的偏导数。

第26行：定义学习算法中参数的调整算法， $\theta_i = \theta_i - \alpha \frac{\partial J}{\partial \theta_i}$ 。

第30行：对所有训练样本，先求出代价函数 J ，然后求代价函数对参数向量的偏导数，最后按照梯度下降算法调整参数值。

完成上述准备工作后，下面就开始讲解真正的线性回归算法训练程序了，代码如下（b011/lr_engine.py 5）：

```
50 def train_lr():
51     print('train linear regression model')
52     for i in range(MAX_STEPS):
53         for j in range(len(trains[0])):
54             nh = train(trains[0][j], trains[1][j])
55             if i % 5 == 0:
56                 print('%d-%d:x=%f h=%f y=%f w=%s t0=%f' % (i, j, trains[0][j]
57                     ], nh, trains[1][j], theta.get_value(), theta0.get_value()))
57     draw_plot()
58     plt.show()
59
```

第52行：循环规定的最大迭代次数。

第53行：对训练集上的每个训练样本执行循环体。

第54行：执行训练函数，首先求出代价函数 J ，然后求代价函数对参数向量的偏导数，

最后按照梯度下降算法调整参数值，并且返回本次的估算值。

第 57、58 行：当执行完最大迭代次数后，打印训练结果。

下面是训练结果绘制程序（b011/lr_engine.py 6），这里只给出了经过平移和缩小之后的规整数据的图形，读者可以运行反变换，得到原始数据，绘制出原始数据的训练结果。

```
33 style.use('fivethirtyeight')
34 fig = plt.figure()
35 ax1 = fig.add_subplot(1,1,1)
36
37 def draw_plot():
38     print('draw plot')
39     ax1.clear()
40     plt.scatter(trains_x, trains[1], label='linear regression',
41               alpha=0.3, edgecolors='none')
42     plt.legend()
43     plt.grid(True)
44     xs = [-0.5, 0.5]
45     v1 = theta.get_value()
46     ys = [-0.5*v1[0][0] + theta0.get_value(),
47          0.5*v1[0][0] + theta0.get_value()]
48     #ys = [300, 2300]
49     ax1.plot(xs,ys)
```

第 33~35 行：定义图形的风格，并生成图形绘制对象。

第 39 行：清空图形内容。

第 40 行：画出模拟训练样本数据的散点。

第 42 行：绘制图例框。

第 43 行：绘制网格线。

第 44 行：进行平移和缩小之后，X 的范围是 $[-0.5, 0.5]$ 。

第 45 行：获取参数值。

第 46 行：由于线性回归模型是一条直线，斜率为参数，所以根据 X 的范围可以算出起点和终点的 Y 轴坐标。

第 49 行：按给定起点坐标和终点坐标，绘制代表线性回归模型的直线。

运行上述程序，可以得到如图 2.6 所示的图形。

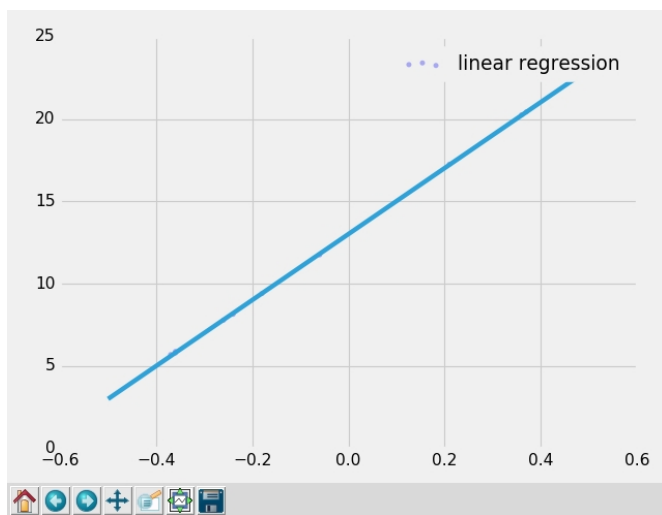


图 2.6 线性回归模型图

由于给定训练数据的误差项很小，所以在图 2.6 中其被我们的模型直线所覆盖，不能显示出来。单独绘制的训练样本如图 2.7 所示。

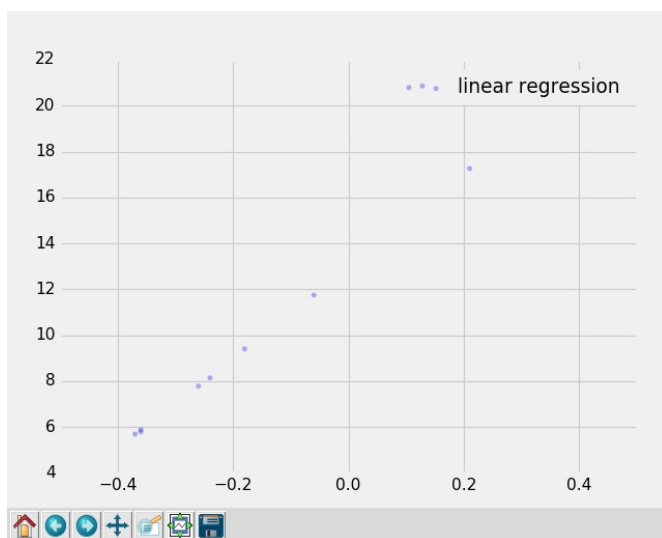


图 2.7 训练数据散点图

由此可以看出，线性回归模型可以很好地拟合我们设计出来的训练样本数据，我们的线性回归算法的实现是正确有效的。

有关线性回归的讨论到此就结束了，读者也许会发现，这部分关于线性回归算法的介绍，有点像斯坦福大学 Andrew NG 教授机器学习算法课程中线性回归算法部分的超详细版，并且原来课程中的实现部分采用的是 Matlib 或 Octavie，而在本书中采用的是 Theano 框架，同时本书将训练样本进行了归一化处理，使其过程更容易收敛。读者如果对机器学习相关内容感兴趣，可以到网易公开课搜索关于机器学习的内容，找到这节课的视频进行学习，课件可以在斯坦福大学课程主页上找到，课程主页的网址为 <http://cs229.stanford.edu/>。这门课程对数学知识的要求很高，所以最好备上高等数学、线性代数和概率论的教材，复习一下相关知识，这样理解起来可能会相对容易一些。这门课程也有些不足，除了更偏重理论，关于神经网络和深度学习的内容也较少，令人有点儿遗憾。

第 3 章

逻辑回归

从本章开始，将正式进入深度学习领域。逻辑回归算法是最简单的一种模式识别算法，虽然其仅能解决线性可分的问题，但是由于其较简单，故当前还在大量使用。例如在医学诊断中，每个疾病都是由一系列症状和综合特征组成的，只要准确地收集每一位患者的症状和综合特征，利用逻辑回归算法就能做出医学专家水准的诊断。由此可见，逻辑回归算法的作用还是相当强大的。

在本章中，将首先研究逻辑回归算法的数学基础，然后以 MNIST 手写数字识别为例，介绍这个数据的格式，以及怎样用逻辑回归算法进行 MNIST 手写数字识别，并且训练模型，使其达到 1% 左右的误差率。

3.1 逻辑回归数学基础

3.1.1 逻辑回归算法的直观解释

逻辑回归算法虽然是非常简单的机器学习算法，但是其数学基础还是比较复杂的。如果刚开始就陷入算法的数学细节中，则很难理解逻辑回归问题的本质，最后只知道一堆数学公式，而不知道具体该怎么将它们应用到实际问题中。这种现象是我们应该着力避免的。

所以先不考虑数学理论，而是讲一下逻辑回归算法的物理意义，使读者对逻辑回归算法有一个直观的了解，为对后续数学理论的理解打下基础。

下面通过一个简单的例子来说明什么是逻辑回归算法。假设在三维空间中有一组待分类的点，同时有一系列平面代表这些点应该属于的类别，我们将通过这些点到代表类别的平面的距离，来判断点属于的类别。也就是说，对于一个点来说，我们找到与其距离最近

的平面，那么就说这个点属于这个类别。上面的讨论是在三维空间下进行的，如果推广到多维空间，那么这里的平面就变成了超平面，但是概念是类似的。

将上述描述转换成数学语言：假设输入向量为 \mathbf{x} ，其维度为 D ，输出类别为 Y ，共有 N 个类别。对于上面的分类问题，如果把问题简化，就变为在二维平面上的点及一系列代表类别的线，求距离该点最近的直线的问题，而直线在二维情况下可以表示为 $y = w\mathbf{x} + b$ ，其中 w 为权重， b 为偏移量。如果将上式推广到高维空间，则权值将变为一个矩阵，偏移量将变为一个向量，可以表示为 $W\mathbf{x} + \mathbf{b}$ ，我们将权值矩阵和偏移量向量称为模型的参数集。

3.1.2 逻辑回归算法数学推导

有了对逻辑回归问题的直观理解之后，开始推导逻辑回归算法。首先讲解单类别逻辑回归问题，如判断患者是否患有某种疾病。然后讨论多类别逻辑回归问题，因为在 MNIST 手写数字识别的例子中，需要判断给定图片是 0~9 中的哪个数字，就是一个多类别模式识别问题。

与线性回归算法类似，下面除了讲述普通迭代法求解，还会讲解利用牛顿法解决逻辑回归问题。通常牛顿法收敛速度更快。

在本章理论部分的最后，会简单讨论一下通用学习模型，因为线性回归和逻辑回归算法都是这种通用学习模型的特例，而且利用通用学习模型，还可以推导出更多机器学习算法。

对于模式分类（Classification）问题，在训练样本中 $y \in \{0,1\}$ ，假设就变为：

$$h_{\theta}(\mathbf{x}) \in [0,1]$$

为了表示假设 $h_{\theta}(\mathbf{x})$ ，使用 Sigmoid 函数，而不是线性回归算法中的线性函数。但选择 Sigmoid 函数并不是随意的，而是有理论基础的。在本节中，先假定是为了方便而选择 Sigmoid 函数来定义假设 $h_{\theta}(\mathbf{x})$ 的。Sigmoid 函数的定义为：

$$g(z) = \frac{1}{1 + e^{-z}}$$

其函数图像如图 3.1 所示（chp03/c001.py）。

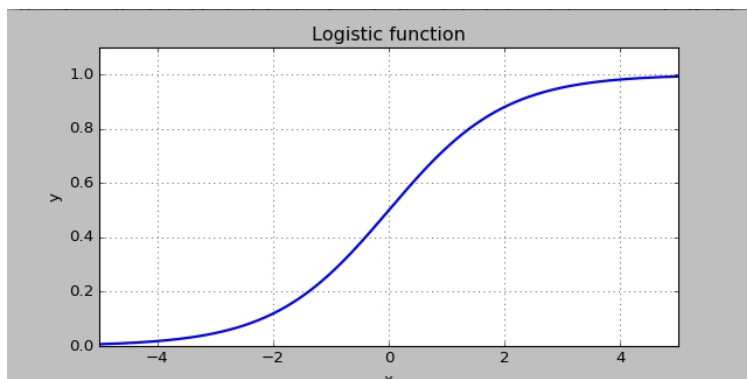


图 3.1 Sigmoid 函数的图像

所以假设可以表示为：

$$h_{\theta}(\mathbf{x}) = g(\theta^T \mathbf{x}) = \frac{1}{1 + e^{-\theta^T \mathbf{x}}}$$

对于 $g(z)$ 这个 Sigmoid 函数，在深度学习算法中通常会用到其导数，下面进行一下推导：

$$\begin{aligned} g'(z) &= \frac{d}{dz} \frac{1}{1 + e^{-z}} = \left(-\frac{1}{(1 + e^{-z})^2} \right) (-e^{-z}) = \frac{e^{-z}}{(1 + e^{-z})^2} = \frac{1}{1 + e^{-z}} \left(\frac{e^{-z}}{1 + e^{-z}} \right) \\ &= \frac{1}{1 + e^{-z}} \left(1 - \frac{1}{1 + e^{-z}} \right) = g(z)(1 - g(z)) \end{aligned}$$

在当前所研究的问题中， y 值只可以取 1 或 0，下面用假设来表示 $y=1$ 和 $y=0$ 时的概率：

$$P(y = 1 | \mathbf{x}; \theta) = h_{\theta}(\mathbf{x})$$

$$P(y = 0 | \mathbf{x}; \theta) = 1 - h_{\theta}(\mathbf{x})$$

实际上，可以进一步简化上面这两个等式并进行合并：

$$P(y | \mathbf{x}; \theta) = h_{\theta}(\mathbf{x})^y (1 - h_{\theta}(\mathbf{x}))^{1-y} \quad (1)$$

上式可以这样理解，当 $y=1$ 时，等式右边第二项的指数值为 0，其值为 1，所以就剩下第一项；当 $y=0$ 时，等式右边第一项的指数值为 0，其值为 1，则只剩下第二项，所以与用两个等式表达的结果相同。

下面来定义似然函数：

$$\mathcal{L}(\theta) = P(y | \mathbf{x}; \theta) = \prod_{i=1}^m P(y^{(i)} | \mathbf{x}^{(i)}; \theta) \quad (2)$$

将式 (1) 代入式 (2) 得：

$$\mathcal{L}(\theta) = \prod_{i=1}^m h_{\theta}(\mathbf{x}^{(i)})^{y^{(i)}} (1 - h_{\theta}(\mathbf{x}^{(i)}))^{1-y^{(i)}}$$

求对数似然函数：

$$\ell(\theta) = \log \mathcal{L}(\theta) = \sum_{i=1}^m \left[y^{(i)} \log h_{\theta}(\mathbf{x}^{(i)}) + (1 - y^{(i)}) \log (1 - h_{\theta}(\mathbf{x}^{(i)})) \right]$$

下面将求 $\ell(\theta)$ 的最大值，也就是求最大对数似然值。与在线性回归算法中求最小值时用到的梯度下降算法类似，在这里采用梯度上升算法：

$$\theta = \theta + \alpha \nabla_{\theta} \ell(\theta) \quad (3)$$

式中， α 为学习率， $\nabla_{\theta} \ell(\theta)$ 表示对数似然函数对参数向量 θ 的偏导。

下面对对数似然函数求相对于 θ_j 的偏导，为了推导方便，将假设 $h_{\theta}(\mathbf{x}^{(i)})$ 用 $g(\theta^T \mathbf{x})$ 来表示：

$$\begin{aligned}
 \frac{\partial \ell(\theta)}{\partial \theta_j} &= \sum_{i=1}^m \left(y^{(i)} \frac{1}{g(\theta^T \mathbf{x})} - (1 - y^{(i)}) \frac{1}{1 - g(\theta^T \mathbf{x})} \right) \frac{\partial}{\partial \theta_j} g(\theta^T \mathbf{x}) \\
 &= \sum_{i=1}^m \left(y^{(i)} \frac{1}{g(\theta^T \mathbf{x})} - (1 - y^{(i)}) \frac{1}{1 - g(\theta^T \mathbf{x})} \right) g(\theta^T \mathbf{x}) (1 - g(\theta^T \mathbf{x})) \frac{\partial}{\partial \theta_j} (\theta^T \mathbf{x}) \\
 &= \sum_{i=1}^m \left(y^{(i)} \frac{1}{g(\theta^T \mathbf{x})} - (1 - y^{(i)}) \frac{1}{1 - g(\theta^T \mathbf{x})} \right) g(\theta^T \mathbf{x}) (1 - g(\theta^T \mathbf{x})) x_j^{(i)} \\
 &= \sum_{i=1}^m \left(\frac{y^{(i)} (1 - g(\theta^T \mathbf{x})) - (1 - y^{(i)}) g(\theta^T \mathbf{x})}{g(\theta^T \mathbf{x}) (1 - g(\theta^T \mathbf{x}))} \right) g(\theta^T \mathbf{x}) (1 - g(\theta^T \mathbf{x})) x_j^{(i)} \\
 &= \sum_{i=1}^m (y^{(i)} - g(\theta^T \mathbf{x})) x_j^{(i)}
 \end{aligned}$$

上面推导的是批量学习时的公式，如果是在线学习，推导过程是一样的，只需要去掉前面的累加符号就可以了。具体推导过程就不在这里讨论了，公式如下：

$$\frac{\partial}{\partial \theta_j} \ell(\theta) = (y^{(i)} - h_{\theta}(\mathbf{x}^{(i)})) x_j^{(i)}$$

我们将上式带入式（3），就可以得到随机梯度上升算法：

$$\theta_j = +\alpha (y^{(i)} - h_{\theta}(\mathbf{x}^{(i)})) x_j^{(i)}$$

上式与线性回归算法中的随机梯度下降算法的学习公式非常相似，那么它们是一类函数吗？答案是否定的。因为在上式中， $h_{\theta}(\mathbf{x}^{(i)})$ 是指数函数而不是线性函数，所以两个公式虽然外形有点儿相似，但却是两种不同的算法。

3.1.3 牛顿法解逻辑回归问题

在线性回归算法中，除了使用迭代算法，还可以使用解析法，即最小二乘法求出参数解。但是在逻辑回归中，假设函数使用的是指数函数，所以很难用解析法求出解。但是还有更好的算法，使收敛速度更快，就是下面要介绍的牛顿法。

先来介绍一下标准的牛顿法，然后再介绍牛顿法在逻辑回归算法中的应用。假设给定一个函数映射 $f: \mathbb{R} \rightarrow \mathbb{R}$ ，我们的任务是发现一点 θ ，使 $f(\theta) = 0$ 。用牛顿法求解这个问题的公式：

$$\theta = \theta - \frac{f(\theta)}{f'(\theta)} \tag{4}$$

只介绍定义会让人感觉很抽象。下面用一个具体的例子来讲解牛顿法的具体算法，使读者对牛顿法有一个直观的认识。

先假设有一个函数：

$$f(x) = x^2 - 2.25 \quad x \in \{x > 0\}$$

我们知道，其与 X 轴的交点为 $x=1.5$ 。假设现在的任务就是给定 $f(x)$ ，以及任意初始值点 $\theta_0 = 4.5$ ，求出 $f(x)=0$ 的坐标，这就是牛顿法要解决的问题。

根据牛顿法，从 θ_0 点开始，在该点做 $f(x)$ 的切线，结果如图 3.2 所示（chp03/c002.py）。

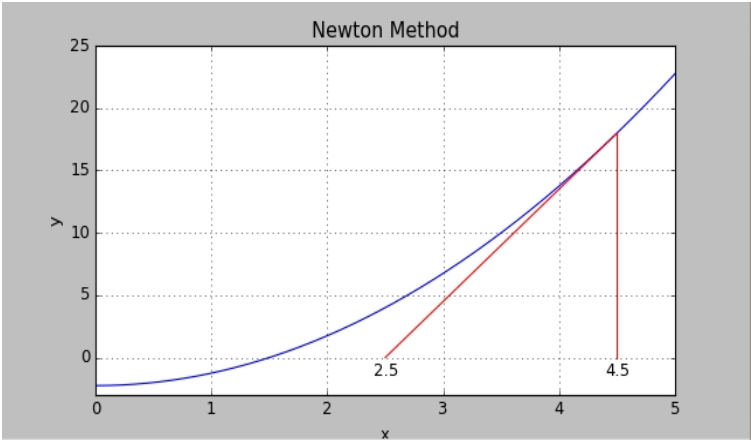


图 3.2 根据牛顿法做切线

根据式（4），可以由 $\theta_0 = 4.5$ 求出第一次迭代 θ_1 的值：

$$\theta_1 = \theta_0 - \frac{f(\theta_0)}{f'(\theta_0)} = \theta_0 - \frac{\theta_0^2 - 2.25}{2\theta_0} = 2.5$$

再从 $\theta_1 = 2.5$ 开始重新执行上面的计算过程，即做切线，与 X 轴交于 θ_2 点，求出绿色线段所表示的部分：

$$\theta_2 = \theta_1 - \frac{f(\theta_1)}{f'(\theta_1)} = \theta_1 - \frac{\theta_1^2 - 2.25}{2\theta_1} = 1.7$$

结果如图 3.3 所示（chp03/c002.py）。

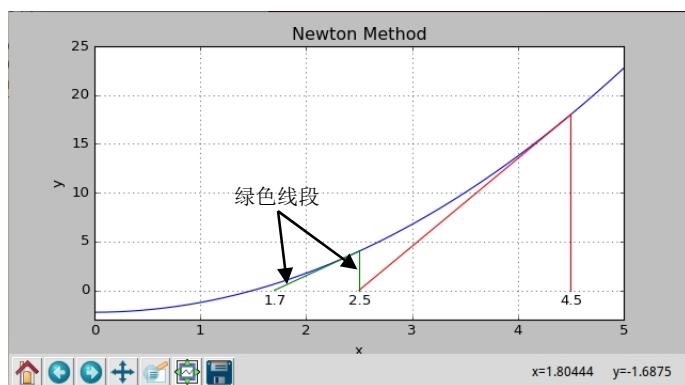


图 3.3 第二次应用牛顿法做切线

此时得到的值即为 1.7，而这个函数与 X 轴交点的真实坐标为 1.5，我们仅通过两次迭代就已经非常接近真实值了。我们相信，如果再连续应用一两次牛顿法，就可以得到 1.5 这个正确值。

从上面的例子可以看出，用牛顿法迭代求值时，可以证明其是以平方的速度向真值点逼近的，如果在离真值点很远的地方，如本次迭代点离真值点的距离为 0.1，那么经过一次迭代之后，其距离将变为 0.01，再经过一次迭代，其距离将进一步缩小为 0.001。由此可见，其收敛速度是相当快的。

下面来讨论怎样将牛顿法应用于逻辑回归问题的求解过程。

在逻辑回归算法中，需要求出对数似然函数 $\ell(\theta)$ 的最大值，根据高等数学知识可以知道，求极值点就等于求对数似然函数导数为 0 的点，如果把 $\ell'(\theta)$ 视为上面讨论中的 $f(x)$ ，则就可以针对 $\ell'(\theta)$ 应用牛顿法，公式为：

$$\theta = \theta - \frac{\ell'(\theta)}{\ell''(\theta)}$$

应用牛顿法可以更快地求出 ℓ 的极值。

在前述公式推导中，并没有强调实际上我们的特征值是 n 维的，所以输入信号为 n 维向量 \mathbf{x} ，参数 θ 也应该是 n 维向量形式，因此我们需要牛顿法的向量形式。

将牛顿法应用于向量形式，可以得到：

$$\theta = \theta - H^{-1} \nabla_{\theta} \ell(\theta)$$

$\nabla_{\theta} \ell(\theta)$ 是对 $\ell(\theta)$ 的每个 θ_i 求偏导，最终组成 \mathbb{R}^n 维向量：

$$\begin{bmatrix} \frac{\partial \ell(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial \ell(\theta)}{\partial \theta_n} \end{bmatrix}$$

式中的 H 为 $\mathbb{R}^{n+1 \times n+1}$ 矩阵，这里引入 $x_0 = 1$ 时的截距项，称为海森矩阵，其定义为：

$$H_{ij} = \frac{\partial^2 \ell(\theta)}{\partial \theta_i \partial \theta_j}$$

通常，在逻辑回归算法中，若特征向量的维度不太高，例如在几百个特征之内，利用牛顿法求解的速度还是相当快的。但是如果特征数量太多，有成千上万维，因为算法需要对海森矩阵求逆，所以运算量会比较大，使用牛顿法就不太合适了，迭代算法可能具有更好的性能。

3.1.4 通用学习模型

到目前为止，我们学习了两种学习算法，分别是线性回归算法和逻辑回归算法，线性回归算法用于解决数值预测问题，逻辑回归算法用于模式分类问题。其实这两种算法都是由所谓的通用线性模型（GLM）派生出来的，而且通用线性模型不仅可以派生出线性回归算法和逻辑回归算法，还可以派生出很多其他的主流算法。

对于线性回归问题，可以将其表示为：

$$P(y|\mathbf{x}; \boldsymbol{\theta}), \quad y \in \mathbb{R} \text{ 且服从高斯分布}$$

上式可以理解为 y 在给定 \mathbf{x} 的情况下，以 $\boldsymbol{\theta}$ 为参数的概率， $(y|\mathbf{x}; \boldsymbol{\theta}) \sim (\boldsymbol{\mu}, \sigma^2)$ 即为高斯分布（Gaussian）。

由概率论的知识可以知道，高斯分布的概率密度函数为：

$$f(\mathbf{x}) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2\sigma^2}(\mathbf{x}-\boldsymbol{\mu})^2}$$

式中， $\boldsymbol{\mu}$ 为期望， σ 为方差。另外，顺便提一下，高斯分布也经常被称为正态分布，其函数图像如图 3.4 所示（c003.py）。

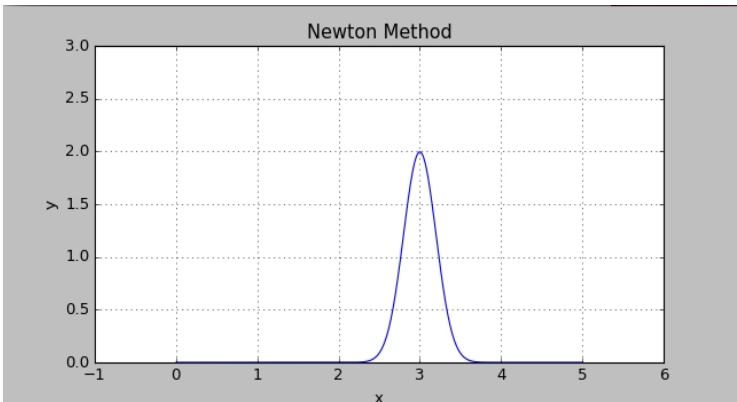


图 3.4 高斯分布概率密度函数

根据以上假设，可以得到线性回归算法，同时可以推导出最小二乘法，这在线性回归算法部分已经详细推导过，这里就不再赘述了。

在逻辑回归问题中，对于 $P(y|\mathbf{x}; \boldsymbol{\theta})$ ，若 $y \in \{0, 1\}$ 且其满足伯努利分布（Bernoulli），即 $\theta \sim \text{Bernoulli}(\theta)$ 分布，可以表示为：

$$P(y = 1|\emptyset) = \emptyset$$

这里需要注意，上式并不表示一个函数，在改变 \emptyset 值时可以得到一系列函数。

同理，对于高斯分布 $N(\mu, \sigma^2)$ ，通过改变 μ 的值也可以得到一系列函数。

下面引入通用线性模型，可以推导出上述两个函数均是这个函数的特殊情况，并且以这个模型为基础，还可以推导出其他有用的模型。

首先定义一个指数族函数：

$$P(y; \eta) = b(y) \exp(\eta^T T(y) - \alpha(\eta)) \quad (5)$$

式中，

- η ：自然参数。
- $T(y)$ ：充分统计量，在通常情况下 $T(y)=y$ 。
- 通过指定不同的 α 、 b 和 T 值，可以得到一族函数，所以称式（5）为指数族函数，这些函数均以 η 为参数。

再来看怎样通过通用线性模型推导得到逻辑回归模型，对于伯努利分布：

$$\text{Bernoulli}(\emptyset): P(y = 1|\emptyset) = \emptyset$$

对于 $y \in \{0,1\}$ 来说，其概率质量函数可以表示为：

$$\begin{aligned} P(y|\emptyset) &= \emptyset^y (1 - \emptyset)^{1-y} = \exp(\log(\emptyset^y (1 - \emptyset)^{1-y})) \\ &= \exp(y \log \emptyset + (1 - y) \log(1 - \emptyset)) \\ &= \exp\left(y(\log \emptyset - \log(1 - \emptyset)) + \log(1 - \emptyset)\right) \\ &= \exp\left(\left(\log \frac{\emptyset}{1 - \emptyset}\right) y + \log(1 - \emptyset)\right) \end{aligned} \quad (6)$$

假设：

- $b(y)=1$ 。
- $\eta = \log \frac{\emptyset}{1 - \emptyset}$ 。
- $T(y)=y$ 。
- $\alpha(\eta) = -\log(1 - \emptyset)$ 。

则式（6）就可以变为通用线性模型。

另外，在伯努利分布下，自然参数 η 的值可以定义为：

$$\eta = \log \frac{\emptyset}{1 - \emptyset}$$

可以求出 \emptyset 值：

$$\emptyset = \frac{1}{1 + e^{-\eta}}$$

上式正是逻辑回归算法中假设 $h_{\theta}(x)$ 的函数形式，这也说明了在逻辑回归算法中选择的 $h_{\theta}(x)$ 是有理论依据的。

下面来看高斯分布，其概率密度函数可以表示为：

$$\begin{aligned} p(y; \mu, \sigma^2) &= \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y - \mu)^2}{2\sigma^2}\right) \\ &= \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{y^2 - 2y\mu + \mu^2}{2\sigma^2}\right) \\ &= \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{y^2}{2\sigma^2}\right) \exp\left(\frac{2y\mu - \mu^2}{2\sigma^2}\right) \\ &= \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{y^2}{2\sigma^2}\right) \exp\left(\frac{\mu y}{\sigma^2} - \frac{\mu^2}{2\sigma^2}\right) \end{aligned}$$

假设：

- ☐ $\eta = \mu$ 。
- ☐ $T(y) = y$ 。
- ☐ $\alpha(\eta) = \frac{\mu^2}{2\sigma^2} = \frac{\eta^2}{2\sigma^2}$ 。
- ☐ $b(y) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{y^2}{2\sigma^2}\right)$ 。

通过指定 α 、 b 、 T 和 η ，可以看出，高斯分布也是一类特殊的通用线性模型。

通过这种指数族分布，实际上可以得到很多分布。

多项式（Multinomial）分布：例如在手写数字识别中，分为 0~9 共 10 种类别，其中只有一个类别为 1，其余类别均为 0。

泊松分布（Poisson）：通常用来估算数量，如估算网站访问量和点击率等，也属于指数族分布。

伽玛分布（Gamma）：通常用于对时间间隔的建模，也是指数族分布的一种。

在讲述了通用线性模型之后，我们自然要问，怎样使用它呢？假设我们有一个微信公众号，想估算文章主题、更新频率、大 V 数量等参数对粉丝数增长的影响。对于粉丝数增长的问题，可以选用泊松分布，因此下面的任务就是设计一种泊松回归算法来解粉丝数增长估算问题。如果以通用线性模型为指导，通过选择恰当的参数，就可以很方便地得出相应的回归算法。

因为在本章的例子中，在使用 MNIST 手写数字识别的实例中，需要用到多项式分布情况，即输出类别为 0~9 共 10 个类别，采用 softmax 函数形式，所以下面就来推导一下多项式分布下的 softmax 函数。

根据通用线性模型做具体的算法设计前，需要有以下 3 个前提条件。

（1） $(y|x; \theta) \sim \exp_family(\eta)$ ，对于变量 y 在给定输入变量 x 和参数 θ 的情况下，其符合以 η 为参数的指数族分布。

（2）我们的目标是在给定 x 的情况下，求出 $T(y)$ 的期望作为假设函数，即 $h_{\theta}(x) =$

$E[T(y)|x] = E[y|x]$ ，因为通常情况下 $T(y)=y$ 。以逻辑回归算法为例，假设 $h_{\theta}(x) = P(y = 1|x; \theta) = 0 \cdot P(y = 0|x; \theta) + 1 \cdot P(y = 1|x; \theta) = E[y|x; \theta]$ 。

(3) 假设自然参数 $\eta = \theta^T x$ ，如果是多维情况， η 为向量形式：

$$\eta = \begin{bmatrix} \theta_1^T x \\ \dots \\ \theta_i^T x \\ \dots \\ \theta_n^T x \end{bmatrix}$$

1. 线性回归问题

对于线性回归问题，我们假设 $(y|x; \theta) \sim N(\mu, \sigma^2)$ 的高斯分布（Gaussian），根据前面的推导，如果把高斯分布问题视为通用线性模型的指数族函数，则其参数如下：

$$P(y; \eta) = b(y) \exp(\eta^T T(y) - \alpha(\eta))$$

$$\eta = \mu$$

$$T(y) = y$$

$$\alpha(\eta) = \frac{\mu^2}{2\sigma^2} = \frac{\eta^2}{2\sigma^2}$$

$$b(y) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{y^2}{2\sigma^2}\right)$$

根据前提条件中的第 2 条得到：

$$h_{\theta}(x) = E[y|x; \theta] = \mu \quad \# \text{因为其是高斯分布的均值}$$

$$= \eta \quad \# \text{上面参数选择}$$

$$= \theta^T x \quad \# \text{第 3 个前提条件}$$

上式的最后一行就是线性回归模型中假设 $h_{\theta}(x)$ 的表达式，由此可见，可以从通用线性模型的指数族函数非常容易地推导出线性回归模型。

2. 逻辑回归问题

逻辑回归问题是一个模式分类问题， $y \in \{0,1\}$ ，只能取 0 或 1 两个值，其背后是伯努利分布：

$$P(y|\theta) = \theta^y (1 - \theta)^{1-y}$$

根据前面的推导过程，对通用线性模型的参数取如下值，可以得到伯努利分布：

$$b(y)=1$$

$$\eta = \log \frac{\theta}{1-\theta} \quad \# * 1$$

$$T(y)=y$$

$$\alpha(\boldsymbol{\eta}) = -\log(1 - \phi)$$

根据第 2 个前提条件可得：

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = P(y = 1|\boldsymbol{x}; \boldsymbol{\theta}) = 0 \cdot P(y = 0|\boldsymbol{x}; \boldsymbol{\theta}) + 1 \cdot P(y = 1|\boldsymbol{x}; \boldsymbol{\theta}) = E[y|\boldsymbol{x}; \boldsymbol{\theta}]$$

因为根据伯努利分布的性质，其均值为 ϕ ，所以可得：

$$\begin{aligned} h_{\boldsymbol{\theta}}(\boldsymbol{x}) &= E[y|\boldsymbol{x}; \boldsymbol{\theta}] = \phi \\ &= \frac{1}{1 + e^{-\boldsymbol{\eta}}} \quad \# \text{由 * 1 式解出 } \boldsymbol{\eta} \\ &= \frac{1}{1 + e^{-\boldsymbol{\theta}^T \boldsymbol{x}}} \quad \# \text{根据第 3 个前提条件} \end{aligned}$$

这正是逻辑回归模型的假设 $h_{\boldsymbol{\theta}}(\boldsymbol{x})$ 的形式，由此可见当初所选择的假设是有理论依据的。

3. 多项式分布

在本章的例子中，将要解决 MNIST 手写数字识别问题，其有 0~9 共 10 类，而上面对逻辑回归问题的讨论只涉及了两个类别的分类问题，所以在本部分我们将讨论多个类别的模式分类问题。

在多类别模式分类问题中， $y \in \{1, 2, \dots, k\}$ ，对于要研究的手写数字识别问题，这里 $k=10$ ，设定每个类别发生的概率为 ϕ_k ，将得到一组概率 $\phi_1, \phi_2, \dots, \phi_k$ 。因为我们处理的是分类问题，所以所有类别出现的概率加在一起应该为 1，即：

$$\sum_{i=1}^k \phi_i = 1$$

所以 $\phi_1, \phi_2, \dots, \phi_k$ 并不是互相独立的，而是有冗余的。原因很简单，我们以第 k 个类别为例，假设：

$$P(y = i|\phi_i) = \phi_i, \quad i \in \{1, 2, \dots, k\}$$

则 ϕ_k 可以由前面的 $k-1$ 个概率表示出来：

$$\phi_k = 1 - \sum_{i=1}^{k-1} \phi_i$$

为了能够从通用线性模型推导出多项式分布，定义 $T(y)$ 为 $k-1$ 维向量（在这里就不等于 y 了！），其定义如下：

$$T(1) = \begin{bmatrix} 1 \\ 0 \\ \dots \\ 0 \\ 0 \end{bmatrix} \quad T(2) = \begin{bmatrix} 0 \\ 1 \\ \dots \\ 0 \\ 0 \end{bmatrix} \quad \dots \quad T(k-1) = \begin{bmatrix} 0 \\ 0 \\ \dots \\ 1 \\ 0 \end{bmatrix} \quad T(k) = \begin{bmatrix} 0 \\ 0 \\ \dots \\ 0 \\ 1 \end{bmatrix}$$

下面定义指示函数 $1\{\cdot\}$ ，当参数值为 True 时，指示函数的值为 1；当参数值为 False 时，指示函数的值为 0，即 $1\{\text{True}\}=1$ 、 $1\{\text{False}\}=0$ 。

利用指示函数，我们可以把多项式分布的 $T(y)$ 表示为：

$$(T(y))_i = 1\{y = i\}$$

下面求当类别为 i 时，多项式分布的期望：

$$E[(T(y))_i] = \sum_{i=1}^k y_i P(y = i) = P(y = i) = \phi_i$$

根据多项式分布的定义，其概率密度函数为：

$$p(y; \phi) = \phi_1^{1\{y=1\}} \phi_2^{1\{y=2\}} \phi_3^{1\{y=3\}} \dots \phi_k^{1\{y=k\}} \quad \# \text{步骤 1}$$

$$= \phi_1^{1\{y=1\}} \phi_2^{1\{y=2\}} \phi_3^{1\{y=3\}} \dots \phi_{k-1}^{1\{y=k-1\}} \phi_k^{1-\sum_{i=1}^{k-1} (T(y))_i} \quad \# \text{步骤 2}$$

$$= \phi_1^{(T(y))_1} \phi_2^{(T(y))_2} \phi_3^{(T(y))_3} \dots \phi_{k-1}^{(T(y))_{k-1}} \phi_k^{1-\sum_{i=1}^{k-1} (T(y))_i} \quad \# \text{步骤 3}$$

$$\begin{aligned} &= \exp \left((T(y))_1 \log(\phi_1) + (T(y))_2 \log(\phi_2) + \dots + \left(1 - \sum_{i=1}^{k-1} (T(y))_i \right) \log(\phi_k) \right) \quad \# \text{步骤 4} \\ &= \exp \left((T(y))_1 \log \left(\frac{\phi_1}{\phi_k} \right) + (T(y))_2 \log \left(\frac{\phi_2}{\phi_k} \right) + \dots + (T(y))_{k-1} \log \left(\frac{\phi_{k-1}}{\phi_k} \right) + \log(\phi_k) \right) \\ &= b(y) \exp(\eta^T T(y) - \alpha(\eta)) \end{aligned}$$

步骤 4 对步骤 1 的值先取对数再取指数，其值不变。两数相乘取对数，等于两数对数相乘，所以步骤 4 中变为 k 个对数值相加，一个数取指数后取对数等于指数乘以该数的对数。

所以有参数取值为：

$$b(y) = 1$$

$$\boldsymbol{\eta} = \begin{bmatrix} \log\left(\frac{\phi_1}{\phi_k}\right) \\ \log\left(\frac{\phi_2}{\phi_k}\right) \\ \dots \\ \log\left(\frac{\phi_{k-1}}{\phi_k}\right) \end{bmatrix}$$

$$\alpha(\boldsymbol{\eta}) = -\log(\phi_k)$$

将参数取上面的值之后，就可以从通用线性模型的指数族函数中得到多项式分布的表示形式了。

对于 $i=1,2,\dots,k$ ，有自然参数的每个分量，即连接函数，可以表示为：

$$\eta_i = \log\left(\frac{\phi_i}{\phi_k}\right) \quad (7)$$

当 $i=k$ 时：

$$\eta_k = \log\left(\frac{\phi_k}{\phi_k}\right) = \log 1 = 0$$

下面要求出 ϕ_i 的表达式，也就是求响应函数，对式（7）两边取指数得：

$$e^{\eta_i} = \frac{\phi_i}{\phi_k}, \quad \phi_i = \phi_k e^{\eta_i} \quad (8)$$

对 k 个类别求合，则有：

$$\sum_{i=1}^k \phi_i = 1 = \phi_k \sum_{i=1}^k e^{\eta_i}$$

由上式可以求出：

$$\phi_k = \frac{1}{\sum_{i=1}^k e^{\eta_i}}$$

将上式代入式（8）可得：

$$\phi_i = \frac{e^{\eta_i}}{\sum_{i=1}^k e^{\eta_i}}$$

上式就是多类别模式分类问题中输出层经常使用的 softmax 函数。在本章及之后各章的 MNIST 手写数字识别实例中，输出层均采用这种形式。

有了上述这些准备工作之后，就可以推导出 softmax 回归模型了，其实这个模型是逻辑回归模型的泛化形式。

因为此时 $\boldsymbol{\eta}$ 为向量模式，其分量定义为：

$$\eta_i = \boldsymbol{\theta}_i^T \mathbf{x}, \quad i = 1, 2, \dots, k-1, \quad \boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \dots, \boldsymbol{\theta}_{k-1} \in \mathbb{R}^{n+1} \text{ 是 } n+1 \text{ 维向量, } n \text{ 为特征数量。}$$

为了便于规整表示，可以令 $\theta_k = \mathbf{0}$ ，即 0 向量，则对应的 η_k 可以表示为 0：

$$\eta_k = \theta_k^T \mathbf{x} = \mathbf{0} \mathbf{x} = 0$$

所以多项式分布的概率密度函数可以表示为：

$$p(y = i | \mathbf{x}; \boldsymbol{\theta}) = \phi_i = \frac{e^{\eta_i}}{\sum_{j=1}^k e^{\eta_j}} = \frac{e^{\theta_i^T \mathbf{x}}}{\sum_{j=1}^k e^{\theta_j^T \mathbf{x}}}$$

假设函数可以表示为：

$$h_{\boldsymbol{\theta}}(\mathbf{x}) = E[T(y) | \mathbf{x}; \boldsymbol{\theta}] = E \begin{bmatrix} 1\{y=1\} \\ 1\{y=2\} \\ \dots \\ 1\{y=k-1\} \end{bmatrix}_{\mathbf{x}; \boldsymbol{\theta}} = \begin{bmatrix} \phi_1 \\ \phi_2 \\ \dots \\ \phi_{k-1} \end{bmatrix} = \begin{bmatrix} \frac{e^{\theta_1^T \mathbf{x}}}{\sum_{j=1}^k e^{\theta_j^T \mathbf{x}}} \\ \frac{e^{\theta_2^T \mathbf{x}}}{\sum_{j=1}^k e^{\theta_j^T \mathbf{x}}} \\ \dots \\ \frac{e^{\theta_{k-1}^T \mathbf{x}}}{\sum_{j=1}^k e^{\theta_j^T \mathbf{x}}} \end{bmatrix}$$

上式中只有 $k-1$ 项，第 k 项的计算可以采用：

$$p(y = k | \mathbf{x}; \boldsymbol{\theta}) = 1 - \sum_{i=1}^{k-1} \phi_i = \frac{e^{\theta_k^T \mathbf{x}}}{\sum_{j=1}^k e^{\theta_j^T \mathbf{x}}}$$

下面来看 softmax 回归算法的学习算法，任务是给定训练样本集 $\{\mathbf{x}^{(i)}, y^{(i)}; i = 1, 2, \dots, m\}$ ，需要求出参数向量 $\boldsymbol{\theta}$ ，使代价函数最小。

首先定义 softmax 回归算法中的对数似然函数：

$$\ell(\boldsymbol{\theta}) = \log \left(\prod_{i=1}^m p(y^{(i)} | \mathbf{x}^{(i)}; \boldsymbol{\theta}) \right) = \sum_{i=1}^m \log(p(y^{(i)} | \mathbf{x}^{(i)}; \boldsymbol{\theta})) = \sum_{i=1}^m \log \left(\prod_{l=1}^k \frac{e^{\theta_l^T \mathbf{x}^{(i)}}}{\sum_{j=1}^k e^{\theta_j^T \mathbf{x}^{(i)}}} \right)^{1\{y=l\}}$$

下面就可以通过梯度上升算法或牛顿法，求出最大对数似然函数所对应的参数向量的解，具体方法与线性回归算法和逻辑回归算法相同。

3.2 逻辑回归算法简单应用

在明白了逻辑回归算法的基本数学原理之后，就可以用逻辑回归算法来解决实际问题。在本节中，将用逻辑回归模型来预测二维平面上两个点集的分类问题，如图 3.5 所示。

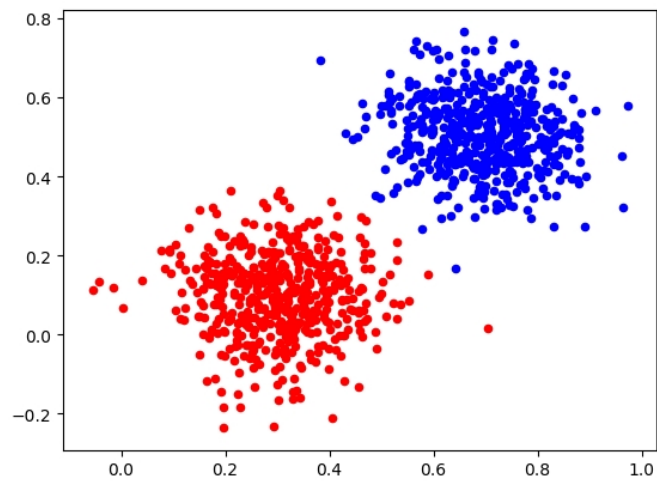


图 3.5 逻辑回归分类问题

如上图所示，我们要研究的问题有两个特征，分别为 x_1 和 x_2 ，有两个类别，我们可以假设图中左下角的点集表示某疾病检测为阳性，而右上角的点集表示疾病检测为阴性。我们的任务就是通过逻辑回归算法，找出一个分类平面： $w_1x_1 + w_2x_2 + b = 0$ ，由于我们研究的是 $n=2$ ，即二维问题，因此分类平面实际上是平面上的一条直线，如图 3.6 所示。

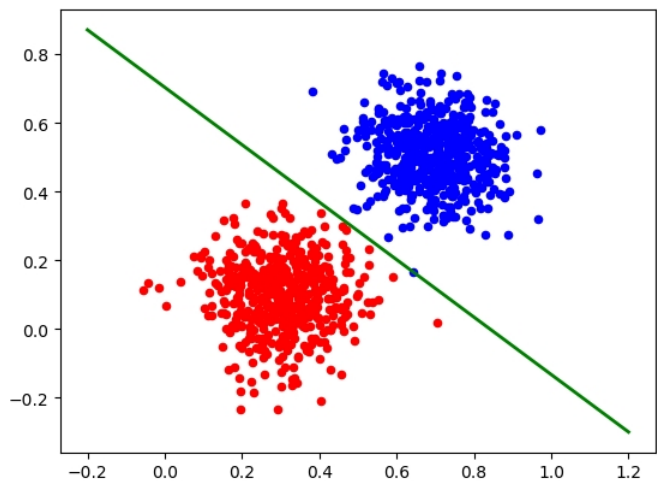


图 3.6 逻辑回归分类平面

我们的任务就是找到图中的直线所代表的平面，也就是找到 $w_1x_1 + w_2x_2 + b = 0$ 中各个系数的值。

在做机器学习算法时，第一步是载入所需要的训练数据。由于这个问题是我们假想的问题，所以需要产生我们的训练数据，并载入到程序中。我们可以自己产生实验数据，但是为了更好的效果，在这里使用从网上下载的数据集，格式如下：

```

1 0,0.147562141324833,0.243518270820358
2 0,0.179868989766322,0.0922537025547999
3 1,0.754244045840797,0.52387485552728
4 0,0.248663780798734,0.175587276306351
5 0,0.397217489998824,0.0342134948381493
6 0,0.45098160780959,0.0328858982745571
7 0,0.335532917252522,0.16654442982869
8 0,0.371372049777255,0.167201755443297
9 0,0.280985655458144,0.214982885821991
10 0,0.313304894476342,0.00521976760984659
11 1,0.638465839375771,0.59044662666132

```

如上所示，数据集文件为一个以逗号分隔的 CSV 文件，第一列 0 代表第一个类别，1 代表第二个类别。第二列和第三列分别代表 x_1 、 x_2 的值。下面我们定义 Seg_Ds_Loader 类来从文件中读出数据，并返回对应的训练样本集、验证样本集和测试样本集，代码如下：

```

1 def prepare_datasets(self, dataset_file, test_file):
2     ''' 将 train_file 分为 train_ds 和 validation_ds 两个文件，分别占 80%和 20%，
3     且为随机分配，test_file 为测试数据集
4     '''
5     train_file = 'datasets/train.csv'
6     validation_file = 'datasets/validation.csv'
7     train_reader = csv.reader(open(dataset_file, encoding='utf-8'))
8     train_writer = csv.writer(open(train_file, 'w', newline=''))
9     validation_writer = csv.writer(open(validation_file, 'w', newline=''))
10    for row in train_reader:
11        rand_num = np.random.uniform(0.0, 1.0, 1)
12        item = [x for x in row]
13        if rand_num[0] > 0.12:
14            train_writer.writerow(row)
15        else:
16            validation_writer.writerow(row)
17    return train_file, validation_file, test_file
18

```

第 1 行：dataset_file 为原始训练样本文件，根据机器学习算法，我们将这部分数据集分为训练样本集和验证样本集。

第 5 行：指定保存所有训练样本集的文件。

第 6 行：指定保存所有验证样本集的文件，验证样本集主要用于避免机器学习算法的过拟合，通常与 Early Stopping 算法一起使用。

第 7 行：使用 csv.reader 打开原始训练样本集。

第 8 行：定义最终训练样本集的 csv.writer，用于写最终的训练样本集文件。注意：这里通过文本方式以写方式打开样本集文件，行尾不加 CR 控制字符（否则会出现隔一行写的情况）。

第 9 行：定义验证样本集的 csv.writer，用于写验证样本集文件。

第 10 行：对原始样本集的每一行循环第 11~16 行操作。

第 11 行：生成一个 0~1 的随机数。

第 12 行：对原始数据集进行必要的预处理（这里没进行任何处理，只是简单地将原始数据进行了复制）。

第 13~16 行：如果上面生成的随机数大于 0.12，则写入训练样本集文件；否则，写入验证样本集文件。这样基本上是 12% 的样本作为验证样本集，88% 的样本作为训练样本集。

由于这个问题比较简单，所以我们的验证样本集偏少，通常将 20%左右的原始样本作为验证样本集。

第 17 行：返回训练样本集文件、验证样本集文件、测试样本集文件。

接下来我们要做的是给定一个样本集文件，读出其中的输入信号特征向量集和输出信号标签集，对于输入信号特征向量，假设其有 m 个样本，特征向量为 n 维，读出的输入信号特征向量集为设计矩阵 $X \in R^{m \times n}$ ，即每一行代表一个样本。代码如下：

```
1 def load_dataset(self, filename, num_labels):
2     ''' 从数据集文件中读出数据集设计矩阵 x 和标签集 y
3     '''
4     X = []
5     y_ = []
6     # 从训练样本集中读出训练样本和标签
7     ds_reader = csv.reader(open(filename, encoding='utf-8'))
8     for row in ds_reader:
9         y_.append(int(row[0]))
10        X.append([float(x) for x in row[1:]])
11    # 将特征变为矩阵
12    X = np.matrix(X).astype(np.float32)
13    # 将标签变为数组，并转化为[1,0]或[0,1]形式，分别代表第一类和第二类
14    y_np = np.array(y_).astype(dtype=np.uint8)
15    y = (np.arange(num_labels) == y_np[:, None]).astype(np.float32)
16    # 返回数据集，以设计矩阵和结果标签形式，形状为：m*n, m*class_num
17    return X, y
```

第 4 行：保存样本的设计矩阵 $X \in R^{m \times n}$ ，其中 m 为样本数， $n=2$ 为特征向量维度，即每一行一个样本。

第 5 行：定义标签集 $y \in R^{m \times c}$ ，其中 m 为样本数， $c=2$ 为类别数，在本例中分别代表阴性和阳性。

第 7 行：通过 `csv.reader` 以 `utf-8` 编码打开样本集文件。

第 8 行：对样本集中的每一行循环第 9、10 行操作。

第 9 行：将本行第一列类别编号加入到 y 列表。

第 10 行：将第二列到最后一列以数组形式加入到 y 列表。

第 12 行：将列表 X 转变为 `numpy` 矩阵类型。

第 14 行：将 $y_$ 变为 y_{np} 数组，值为 `[1, 0, 1, 1, 0, ...]`。

第 15 行：将其变为向量形式 y ，值为 `[[0,1], [1,0], [0,1], [0,1], [1,0], ...]`。

第 17 行：返回设计矩阵 $X \in R^{m \times n}$ 和标签集 $y \in R^{m \times c}$ 。

定义了数据集载入工具类之后，当前的主要任务是设计逻辑回归引擎类 `Lgr_Engine`，用来定义模型的计算图、代价函数和优化算法。

首先来看数据集载入方法，代码如下：

```
1 def load_datasets(self):
2     loader = Seg_Ds_Loader()
3     train_file, validation_file, test_file = loader.prepare_datesets(
4         self.datasets_file, self.test_file)
5     num_labels = 2
6     X_train, y_train = loader.load_dataset(
```

```

7         train_file, num_labels)
8     X_validation, y_validation = loader.load_dataset(
9         validation_file, num_labels)
10    X_test, y_test = loader.load_dataset(
11        test_file, num_labels)
12    return X_train, y_train, X_validation, \
13        y_validation, X_test, y_test, X_train.shape[0]

```

第2行：初始化数据集载入类 `Seg_Ds_Loader`。

第3、4行：生成训练样本集、验证样本集、测试样本集文件，其中训练样本集和测试样本集是从原始训练数据集中按88%和12%分割而成的。

第5行：分类别，在本例中为2，代表阴性和阳性。

第6、7行：从训练样本集文件中读出特征向量的设计矩阵 $X_{\text{train}} \in \mathbb{R}^{m \times n}$ ，其中 m 为样本数量， $n=2$ 为特征维度；标签集 $y_{\text{train}} \in \mathbb{R}^{m \times c}$ ，其中 $c=2$ 为类别数。

第8、9行：从验证样本集文件中读出特征向量的设计矩阵 $X_{\text{validation}} \in \mathbb{R}^{m \times n}$ ，其中 m 为样本数量， $n=2$ 为特征维度；标签集 $y_{\text{validation}} \in \mathbb{R}^{m \times c}$ ，其中 $c=2$ 为类别数。

第10、11行：从测试样本集文件中读出特征向量的设计矩阵 $X_{\text{test}} \in \mathbb{R}^{m \times n}$ ，其中 m 为样本数量， $n=2$ 为特征维度；标签集 $y_{\text{test}} \in \mathbb{R}^{m \times c}$ ，其中 $c=2$ 为类别数。

第12、13行：返回训练样本集、验证样本集、测试样本集。

载入训练样本集、验证样本集、测试样本集之后，再来看模型构建方法 `build_model`，代码如下：

```

1 def build_model(self):
2     X = tf.placeholder("float", shape=[None, self.n])
3     y_ = tf.placeholder("float", shape=[None, self.c])
4     W = tf.Variable(tf.zeros([self.n, self.c]))
5     b = tf.Variable(tf.zeros([self.c]))
6     z = tf.matmul(X, W) + b
7     y = tf.nn.softmax(tf.matmul(X, W) + b)
8     cross_entropy = -tf.reduce_sum(y_*tf.log(y)) # 效果好，仅需5次迭代
9     #cross_entropy = tf.reduce_sum(tf.pow(y_-y, 2)) / (2*m) # 需要
10    # 10000 次左右迭代才能得到有意义的结果
11    train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)
12    # 评估模型
13    predicted_class = tf.argmax(y, 1);
14    correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
15    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
16    return X, y_, W, b, y, cross_entropy, train_step, \
17        predicted_class, correct_prediction, accuracy;

```

第2行：定义输入信号的 `placeholder`，在 TensorFlow 中，在输入信号和输出信号的每次信号前向传播和误差反向传输中，其值都会改变，但是我们不需要学习算法来优化其中的值，因此把它们定义为 `placeholder`，就是占位符。其形状为 `[None, 特征向量维度]`，其中第1维是 `None`，表明在运行时才会赋上具体的值。在运行时，我们基本都是基于迷你批次的随机梯度下降算法（SGD），这时系统自动将 `None` 变为迷你批次中的样本数，每行代表一个样本数据。

第3行：定义计算出的输出信号的 `placeholder`，形状为 `[None, 类别数]`，与输入信号相

似，在运行时，系统自动将第 1 维 None 变为迷你批次中的样本数，每行代表该样本对应的分类类别。每行为一个 c 维 one-hot 向量，在本例中为阴性和阳性两类，所以每行只能为 [1, 0] 或 [0, 1]。

第 4 行：定义输入层到输出层的连接权值矩阵，形状为 [特征向量维度，分类类别数]。

第 5 行：定义输出层的 bias 向量，形状为 [分类类别数]。

第 6 行：定义输入层向输出层传输的信号。

第 7 行：首先计算输出层的输入信号 z ，然后对 z 应用 softmax 函数：

$$y_i = \frac{e^{x_i w_i + b_i}}{\sum_{j=1}^c e^{x_j w_j + b_j}}$$

第 8 行：定义基于交叉熵的代价函数，由于我们研究的问题是一个二分类问题，因此 y_i 符合伯努利分布，即：

$$P(y_- = 1 | \mathbf{x}; \boldsymbol{\theta}) = y$$

$$P(y_- = 0 | \mathbf{x}; \boldsymbol{\theta}) = 1 - y$$

式中， y_- 为正确值， y 为模型的计算值。我们可以将两式统一起来：

$$P(y_- | \mathbf{x}; \boldsymbol{\theta}) = y^{y_-} (1 - y)^{1 - y_-}$$

对于训练样本集中的一个样本 i ，似然函数可以定义为：

$$\mathcal{L}(\boldsymbol{\theta}) = P(y_-^{(i)} | \mathbf{x}; \boldsymbol{\theta}) = P(y^{(i)} | \mathbf{x}^{(i)}; \boldsymbol{\theta})$$

将概率公式代入可得：

$$\mathcal{L}(\boldsymbol{\theta}) = y^{(i) y_-^{(i)}} (1 - y^{(i)})^{1 - y_-^{(i)}}$$

据此得到对数似然函数：

$$\ell(\boldsymbol{\theta}) = -\log \mathcal{L}(\boldsymbol{\theta}) = -y_-^{(i)} \log y^{(i)} - (1 - y_-^{(i)}) \log(1 - y^{(i)})$$

我们知道，当正确值 $y_- = 1$ 时，上式等于 $-y_-^{(i)} \log y^{(i)}$ ，而 $y_- = 0$ 时，计算值 $y^{(i)}$ 也应该趋近于零，因此 $(1 - y_-^{(i)}) \log(1 - y^{(i)})$ 为零，所以可以将代价函数简化为：

$$\ell(\boldsymbol{\theta}) = -\log \mathcal{L}(\boldsymbol{\theta}) = -y_-^{(i)} \log y^{(i)}$$

在上面的推导中，只用到了训练样本集中的一个样本；当我们采用的是迷你批次随机梯度下降算法，所以需要用到 `tf.reduce_sum` 来进行求和。

通过上面的推导，我们就不难理解第 8 行的代码了。

第 9、10 行：定义了基于最小平方差的代价函数，先求计算出来的类别与正确类别之间的误差，然后求平方和。这种形式的代价函数最直观，但是正如注释中所说，这同时是一种低效的代价函数形式，与第 8 行定义的代价函数相比，需进行很多次迭代才能达到预期效果。实践证明，基于最大似然函数或负对数似然函数的代价函数，通常具有更好的效果，一般我们把这种形式的代价函数作为默认的选择。

第 11 行：对交叉熵采用随机梯度下降算法进行优化。

第 13 行：计算出的结果 y 的格式为 $[[0,1],[1,0],[1,0],[0,1],\dots]$ ，我们利用 TensorFlow 的 `argmax` 函数，求出每行最大值的下标，得到如下结果：`predicted_class=[1,0,0,1,\dots]`。

第 14 行：利用 TensorFlow 的 `argmax` 函数，分别求出计算类别向量每个样本的最大值下标和类别向量每个样本的最大值下标，并对其进行比较。这个解释比较抽象，我们来看一个实例，假设经过 TensorFlow 的 `argmax` 函数之后，计算类别结果为： $[1, 0, 0, 1, 0, \dots]$ ，而正确类别结果为： $[1, 0, 1, 1, 0, \dots]$ ，这时运行 TensorFlow 的 `equal` 函数之后的结果为： $[True, True, False, True, True, \dots]$ ，即其为将两个列表逐元素进行比较，相同为 `True`，不同为 `False`，并以一个列表形式返回。

第 15 行：求出预测精度，首先调用 TensorFlow 的 `cast` 函数，将第 14 行的结果变为浮点数列表： $[1.0, 1.0, 0.0, 1.0, 1.0]$ ，这里假设只有 5 个样本。再调用 TensorFlow 的 `reduce_mean` 函数求出这个列表的平均值： $(1.0+1.0+0.0+1.0+1.0)/5=0.8$ ，这个值就是模型预测的精度。

第 16 行：将上述这些值返回给调用者。

下面我们来看 `Lgr_Engine` 类的构造函数：

```
1 import numpy as np
2 import csv
3 import matplotlib.pyplot as plt
4 import tensorflow as tf
5 from seg_ds_loader import Seg_Ds_Loader
6
7 class Lgr_Engine(object):
8     def __init__(self, datasets_file, test_file, n, c, batch_size):
9         self.datasets_file = datasets_file #'datasets/linear_data_train.csv'
10        self.train_file = 'datasets/train.csv'
11        self.test_file = test_file #'datasets/linear_data_eval.csv'
12        self.n = n # 特征向量维度
13        self.c = c # 类别数
14        self.batch_size = batch_size
15
```

第 9 行：定义原始训练数据集文件名称。

第 10 行：定义训练样本集文件名称。

第 11 行：定义测试样本集文件名称。

第 12 行：定义特征向量维度。

第 13 行：定义类别数。

第 14 行：定义迷你批次大小。

接下来我们看类 `Lgr_Engine` 中最重要的方法 `train`，代码如下：

```
1 def train(self, mode=0):
2     ''' mode=0 为全新训练; mode=1 为继续训练 '''
3     X_train, y_train, X_validation, y_validation, X_test, y_test, m = self.load_datasets()
4     X, y_, W, b, y, cross_entropy, train_step, predicted_class, correct_prediction,
5     accuracy = self.build_model()
6     # 装入测试样本
7     test_data_node = tf.constant(X_test)
8     num_epochs = 5
```

```

9     saver = tf.train.Saver()
10    with tf.Session() as sess:
11        sess.run(tf.global_variables_initializer())
12        for epoch in range(num_epochs):
13            batch_num = m // self.batch_size
14            for batch_idx in range(batch_num):
15                offset = batch_idx * self.batch_size
16                X_mb = X_train[offset:(offset+self.batch_size), :]
17                y_mb = y_train[offset:(offset+self.batch_size)]
18                sess.run(train_step, feed_dict={X: X_mb, y_: y_mb})
19            print('W:{0}'.format(sess.run(W)))
20            print('b:{0}'.format(sess.run(b)))
21            print("Accuracy:", accuracy.eval(feed_dict={X: X_test, y_: y_test}))
22            saver.save(sess, 'work/lgr.ckpt')
23            self.plot(sess.run(W), sess.run(b), self.train_file)
24

```

第 3 行：通过 `load_datasets` 方法读入训练样本集、验证样本集、测试样本集。

第 4 行：通过调用本类的 `build_model` 方法，构建基于 TensorFlow 的计算图模型。

第 7 行：将测试样本集数据变为 TensorFlow 的张量。

第 8 行：定义学习整个训练样本集的次数，这里采用的是负对数似然函数的代价函数，如果采用最小平方误差代价函数，这个值应该取大于 10000 的值。

第 9 行：调用 TensorFlow 的 `saver` 来保存模型权值和偏移量等参数数据，或者恢复之前保存的模型参数。

第 10 行：启动 TensorFlow 的会话进程。

第 11 行：初始化 TensorFlow 的变量。

第 12 行：循环第 13~18 行，处理一遍训练样本集。

第 13 行：通过训练样本集样本数除以迷你批量样本大小，得到总共迷你批次数量。

第 14 行：循环第 15~18 行，对所有迷你批次进行处理。

第 15 行：求出本迷你批次的开始位置。

第 16 行：取出本迷你批次的特征向量组成的设计矩阵 `X_mb`。

第 17 行：取出本迷你批次的正确结果标签集 `y_mb`。

第 18 行：执行一次训练操作，包括信号的前向传播，先求出模型输出，然后计算出代价函数值，求出代价函数值导数，再利用随机梯度下降算法进行误差反向传播，修改权值和偏移量等模型参数。

第 19 行：训练完成之后，打印权值向量内容。

第 20 行：打印偏移量内容。

第 21 行：打印在测试样本集中求出的预测精度。

第 22 行：保存模型参数，这里是连接权值矩阵和偏移量。

第 23 行：调用 `plot` 方法，绘制出训练样本集散点图和分割平面（直线）图，使我们对逻辑回归分类问题有一个直观的理解。

下面我们来看具体的绘图函数，代码如下：

```

1 def plot(self, W, b, filename):
2     x01 = []

```

```

3     x02 = []
4     x11 = []
5     x12 = []
6     ds_reader = csv.reader(open(filename, encoding='utf-8'))
7     for row in ds_reader:
8         if int(row[0]) > 0:
9             x11.append(float(row[1]))
10            x12.append(float(row[2]))
11        else:
12            x01.append(float(row[1]))
13            x02.append(float(row[2]))
14    plt.scatter(x01, x02, s=20, color='r')
15    plt.scatter(x11, x12, s=20, color='b')
16
17    w1 = W[0][0]
18    w2 = W[1][0]
19    b_ = b[0]
20    x1 = np.linspace(-0.2, 1.2, 100)
21    x2 = [-(w1/w2)*x-b_/w2 for x in x1]
22    plt.plot(x1, x2, 'g-', label='Plan', linewidth=2)
23
24    plt.show()
25

```

第 2、3 行：定义第一类别样本的横、纵坐标数据。

第 4、5 行：定义第二类别样本的横、纵坐标数据。

第 6 行：创建 `csv.reader` 对象，读取训练样本集数据。

第 7 行：循环第 8~13 行操作，处理训练样本集每个样本。

第 8~10 行：如果是第二类别，加入到第二类别样本的横、纵坐标数据中。

第 11~13 行：如果是第一类别，加入到第一类别样本的横、纵坐标数据中。

第 14 行：绘制第一类别样本的散点图。

第 15 行：绘制第二类别样本的散点图。

第 17~19 行：求出分割平面（直线）的参数。对于这个二分类问题，分类平面（直线）的形式为 $w_1x_1 + w_2x_2 + b = 0$ ，权值 W 为 2×2 的矩阵，共有 4 个元素。如果按照 `Softmax` 分类，假设有 c 个类别，计算 `softmax` 函数时，只需计算 $c-1$ 个类别即可，因为 c 个类别的和为 1，最后一个类别可以使用 1 减去前 $c-1$ 个类别之和得到，为了便于处理，会计算 c 个类别，所以就存在冗余的情况。再回到这个例子，这是一个二分类问题，实际上只需计算一个类别的 `softmax` 函数即可，而这里计算的是两个值，因此是存在冗余的。所以只取 W 的第一列数据（如果取第二列，得到的结果是一样的）。偏移量 b 也取第一个元素，这样就得到了分类平面（直线） $w_1x_1 + w_2x_2 + b = 0$ 的所有参数。

第 20 行：定义横坐标的范围，是 $(-0.2, 1.2)$ ，共有 100 个点。

第 21 行：根据 $x_2 = \frac{w_1}{w_2}x_1 - \frac{b}{w_2}$ ，计算 $(-0.2, 1.2)$ 的 100 个点对应的纵坐标值。

第 22 行：绘制分类平面（直线）。

运行训练算法，大概几十秒就可以得到在测试样本集上 100% 的精度，模型参数如下：

```
W:[[-1.66829371  1.66829371]
 [-1.96494269  1.96494269]]
b:[ 1.3462615  -1.34626126]
Accuracy: 1.0
```

同时绘制出如图 3.7 所示的数据散点图和分类平面（直线）图。

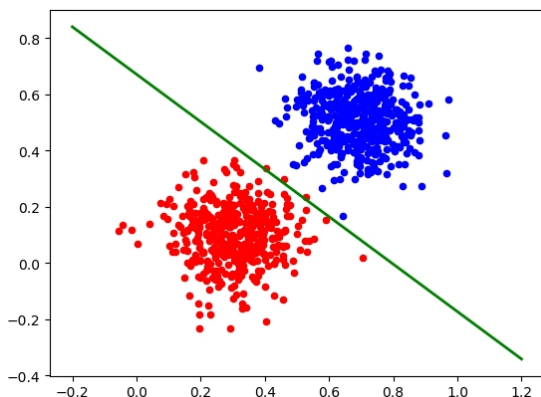


图 3.7 逻辑回归分类直观解释

如图所示，图中左下角的点为第一类别，右上角的点为第二类别，中间的斜线为分类平面（直线），而且，中间的分类平面（直线）可以很好地将两类进行区分。

在模型训练完成之后，就需要将模型置为运行状态，对实际的数据进行预测，实时给出观测样本的类别，下面我们来看一下逻辑回归引擎类 `Lgr_Engine` 的 `run` 方法，代码如下：

```
1 def run(self, x):
2     print('run in run mode')
3     X, y_, W, b, y, cross_entropy, train_step, predicted_class, \
4         correct_prediction, accuracy = self.build_model()
5     init = tf.initialize_all_variables()
6     saver = tf.train.Saver()
7     with tf.Session() as sess:
8         sess.run(init)
9         saver.restore(sess, 'work/lgr.ckpt')
10        y_val = sess.run(y, feed_dict={X: x})
11        print('rst:{0} c1:{1} c2:{2}'.format(y_val,
12        y_val[0][0], y_val[0][1]))
```

第 3、4 行：调用 `build_model` 方法创建 TensorFlow 的计算图。

第 5 行：定义初始化所有变量的方法。

第 6 行：初始化 TensorFlow 的 `saver` 对象，用于恢复之前保存的模型参数。

第 7 行：创建 TensorFlow 会话。

第 8 行：运行变量初始化方法。

第 9 行：恢复训练阶段保存的模型参数。

第 10 行：计算模型输出的 `softmax` 值，即各类别的概率。

第 11 行：打印出学习结果。

运行上面的程序，打印出如下所示的结果。

```
rst:[[ 0.84723103  0.15276901]]
c1:0.8472310304641724 c2:0.15276901423931122
```

从上面的结果可以看出，该样本属于第一类，概率为 84.7%，属于第二类的概率仅为 15.3%，所以可以比较肯定地说，这个样本属于第一类。

完成逻辑回归引擎类 `Lgr_Engine` 之后，我们来看怎样使用这个类，代码如下：

```
1 import numpy as np
2 import csv
3 from lgr_engine import Lgr_Engine
4
5 class Lgr_App:
6     def __init__(self):
7         pass
8
9     def startup(self):
10         lgr_engine = Lgr_Engine('datasets/linear_data_train.csv',
11                                 'datasets/linear_data_eval.csv', 2, 2, 100)
12         #lgr_engine.train()
13         x = [[0.27, 0.02]]
14         lgr_engine.run(x)
15
16 if '__main__' == __name__:
17     lgr_app = Lgr_App()
18     lgr_app.startup()
```

第 10 行：初始化逻辑回归引擎类，参数为：原始训练样本集文件名、测试样本集文件名、特征向量维度、分类类别数、迷你批次大小。

第 12 行：如果是全新训练，直接调用 `train` 方法。

第 13 行：准备一个需要预测的样本。

第 14 行：调用 `run` 方法产生预测结果。

通过上面的讨论可以看出，利用 `TensorFlow` 来实现逻辑回归算法还是比较简单的。我们只需要定义好计算图，无论是信号的前向传播，还是误差的反向传输，都是由 `TensorFlow` 自动完成的，我们不需要关注这些学习算法的细节，降低了深度学习的应用门槛，是工业实现领域首选的深度学习平台技术。

另外，在实际运行部分只写了一个简单的程序，打印出分类结果。本书第 4 章将讨论开发一个 `Web` 服务器，将逻辑回归模型的分类方法作为一个 `Web` 服务向外界提供，方便其他应用程序调用。这样读者就能构建一个完整的深度学习云平台，并将其应用在自己的实际工作中。

3.3 MNIST 手写数字识别库简介

本章的重点是应用逻辑回归算法处理 MNIST 手写数字识别问题。在讲解利用逻辑回归算法处理 MNIST 手写数字识别问题之前，首先来介绍一下 MNIST 手写数字识别样本集的基本情况。

MNIST 是 Mixed National Institute of Standards and Technology 的简称，是国际公认的大型手写数字识别数据集，是机器学习算法验证的标准数据集，其作用有点像生物学研究中的大肠杆菌，可以方便地用于各种机器学习算法间性能的比较。实际上，MNIST 每年都会组织算法竞赛，以检验各种算法的有效性。

MNIST 数据集由纽约大学 Yann LeCun 教授整理推出，每个手写数字图片的大小均为 28×28，黑底白字，并且位于图片中央，共有 60000 个训练样本集和 10000 个测试样本集，测试样本集是不公开的。

MNIST 手写数字图片如图 3.8 所示。

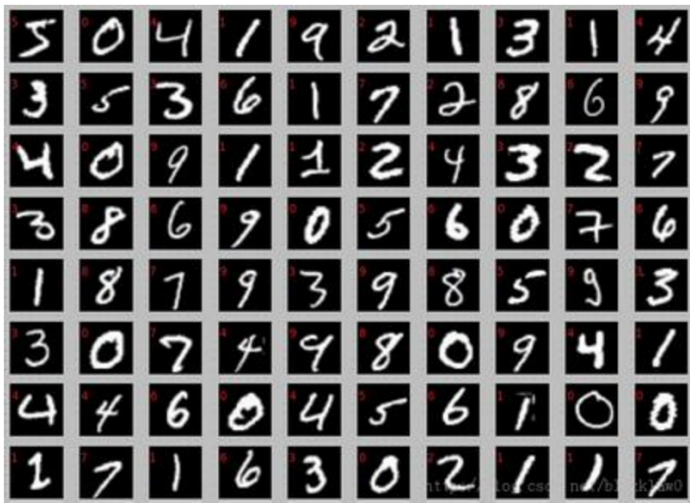


图 3.8 MNIST 手写数字图片示例

图 3.8 只是几张 MNIST 图片的示例。从这些图片可以看出，即使让人来识别，有一些图片也是很不好识别的，但是采用深度学习技术，计算机识别的误差率可以达到 0.23%，优于人的识别结果。这说明人工智能和深度学习技术的发展还是相当振奋人心的。

表 3.1 中列出了典型的机器学习算法在 MNIST 手写数字识别数据集上取得的最好效果。

表 3.1 最佳表现

类 型	分 类 器	变形处理	预 处 理	误差 (%)
线性分类器	线性分类器	否	是	7.6
K最小近邻	K最小近邻，非线性变形	否	是	0.52
Boost Stumpt		否	是	0.87
非线性分类器	40维特征向量，PCA算法	否	否	3.3

续表

类 型	分 类 器	变形处理	预 处 理	误差 (%)
支撑向量机	虚拟SVM	否	是	0.56
神经网络1	2层：784-800-10	否	否	1.6
神经网络2	2层：784-800-10	是	否	0.7
深层网络	6层：784-2500-2000-1500-1000-500-10	否	否	0.35
卷积神经网络1	6层：784-40-80-500-1000-2000-10	是	是	0.31
卷积神经网络2	由35个单独的卷积神经网络组成，每个均为 1-20-P- 40-P-150-10，结果由35个网络投票得出	是	是	0.23

由表 3.1 可以看出，卷积神经网络在 MNIST 数据集上取得了最好的实验效果。表 3.1 中有一列表格表示是否进行了变形处理，从中可以发现，取得较好识别效果的网络基本都进行了变形处理，即对训练样本集数据，通过对图形进行矢量化或拉伸等变形处理，解决了训练样本不足的问题，通过增加训练样本数量，增强了机器学习算法的泛化能力。由于测试样本集不公开，测试样本集上的误差率是衡量机器学习算法的标准。

另外，请注意 6 层深层网络没有进行变形处理，同样取得了 0.35% 的误差率的好成绩，接近单个卷积神经网络的效果，如果加入变形处理，有可能超过卷积神经网络的效果，非常值得关注。有关这个网络的细节，请参考网址 <https://arxiv.org/pdf/1003.0358v1.pdf> 的内容。

本书将涉及表 3.1 中的线性分类器、神经网络 1、卷积神经网络 1 这三种算法，如果读者对其他机器学习算法有兴趣，可以到网址 https://en.wikipedia.org/wiki/MNIST_database#cite_note-19 查看相关文章。

MNIST 手写数字识别数据集由四个文件组成，下载地址为 <http://yann.lecun.com/exdb/mnist/>。文件格式的描述如表 3.2~表 3.6 所示，更加详细的内容也可以参考上面的网址。

表 3.2 MNIST数据集文件描述

文 件 名	说 明
Train-images-idx3-ubyte	训练样本集，图片中每像素的值
Train-labels-idx1-ubyte	训练样本集，每个图片所属类别
T10k-images-idx3-ubyte	测试样本集，图片中每像素的值
T10k-labels-idx1-ubyte	测试样本集，每个图片所属类别

表 3.3 训练样本集图片文件格式描述

Train-images-idx3-ubyte（大端存储）			
文件偏移量	数据类型	值	说 明
0000	Int32	2051	魔术数，表明文件类型为训练样本集图片文件
0004	Int32	60000	训练样本集数量
0008	Int32	28	图片水平分辨率
0012	Int32	28	图片垂直分辨率
0016	byte	...	(0,0)像素点值（0~255）
0017	byte	...	(0,1)像素点值（0~255）
...

表 3.4 训练样本集标签文件格式描述

Train-images-idx3-ubyte（大端存储）			
文件偏移量	数据类型	值	说 明
0000	Int32	2049	魔术数，表明文件类型为训练样本集标签文件
0004	Int32	60000	训练样本集数量
0008	byte	?	第一个图片的类别（0~9）
0009	byte	?	第一个图片的类别（0~9）
...

表 3.5 测试样本集图片文件格式描述

T10k-images-idx3-ubyte（大端存储）			
文件偏移量	数据类型	值	说 明
0000	Int32	2051	魔术数，表明文件类型为测试样本集图片文件
0004	Int32	10000	测试样本集数量
0008	Int32	28	图片水平分辨率
0012	Int32	28	图片垂直分辨率
0016	byte	...	(0,0)像素点值（0~255）
0017	byte	...	(0,1)像素点值（0~255）
...

表 3.6 测试样本集标签文件格式描述

T10k-labels-idx3-ubyte（大端存储）			
文件偏移量	数据类型	值	说 明
0000	Int32	2049	魔术数，表明文件类型为测试样本集标签文件
0004	Int32	60000	测试样本集数量
0008	byte	?	第一个图片的类别（0~9）
0009	byte	?	第一个图片的类别（0~9）
...

从上面的介绍可以看到，MNIST 手写数字识别数据集虽然比较简单，但是直接用 Python 来解析文件还是比较复杂的，如果使用 Python 的 cPickle 库就可以变得比较简单。在 3.4 节使用逻辑回归算法进行手写数字识别的实例中，就将用 cPickle 库对 MNIST 数据集文件进行操作，同时因为 cPickle 库是 Python 的标准序列化库，故模型参数的存储也将使用这个库。

3.4 逻辑回归 MNIST 手写数字识别

在上一节中，我们将逻辑回归算法应用于一个简单的平面点集的分类问题，读者应该已经掌握了 TensorFlow 用于逻辑回归算法实现的基本技巧。但是上一节的例子中的问题比

较简单，我们并没有使用验证样本集和 Early Stopping 等调整技术，而在实际中，这两个方面的应用对提高模型的泛化能力是非常重要的。因此，在本节中，我们通过将逻辑回归算法应用于 MNIST 手写数字识别这一领域，让读者可以看到将 TensorFlow 完整应用于一个实际的模式分类问题所用到的全部技术。

将逻辑回归算法用于 MNIST 手写数字识别，首先需要对问题进行建模。根据 3.3 节 MNIST 手写数字数据集的介绍，对这个问题进行建模。

- ❑ 训练样本集为 $28 \times 28 = 784$ 的黑白图片，每个像素点的值为 $0 \sim 255$ 之间的数，因此输入信号 \mathbf{x} 可以定义为： $\mathbf{x} \in \mathbb{R}^{784}$ 。
- ❑ 模型输出为 $0 \sim 9$ 之间的 10 个类别，所以 $k=10$ 。这就决定输出层需要采用 softmax 回归形式。
- ❑ 训练样本集有 60000 条记录，将其中一部分用于验证数据集，以提高逻辑回归模型的泛化能力。

由于 MNIST 手写数字识别数据集是机器学习领域应用最广泛的数据集，TensorFlow 中已经集成了 MNIST 数据集载入的类，这样研发人员就不用自己来实现载入 MNIST 手写数字识别数据集的方法了。

载入 MNIST 手写数字识别数据集的代码如下：

```

1 import sys
2 import numpy as np
3 import argparse
4 import matplotlib.pyplot as plt
5 import matplotlib.image as mpimg
6 from skimage import io
7 import TensorFlow as tf
8 from TensorFlow.examples.tutorials.mnist import input_data
9 from lgr_engine import Lgr_Engine
10
11 class Lgr_App:
12     def __init__(self):
13         pass
14
15     def learn_mnist(self):
16         mnist = input_data.read_data_sets(FLAGS.data_dir,
17             one_hot=True)
18         X_train = mnist.train.images
19         y_train = mnist.train.labels
20         X_validation = mnist.validation.images
21         y_validation = mnist.validation.labels
22         X_test = mnist.test.images
23         y_test = mnist.test.labels
24         print('X_train: {0} y_train:{1}'.format(
25             X_train.shape, y_train.shape))
26         print('X_validation: {0} y_validation:{1}'.format(
27             X_validation.shape, y_validation.shape))
28         print('X_test: {0} y_test:{1}'.format(
29             X_test.shape, y_test.shape))
30         image_raw = (X_train[1] * 255).astype(int)
31         image = image_raw.reshape(28, 28)
32         label = y_train[1]
```

```

33     idx = 0
34     for item in label:
35         if 1 == item:
36             break
37         idx += 1
38     plt.title('digit:{0}'.format(idx))
39     plt.imshow(image, cmap='gray')
40     plt.show()

```

第 15 行：定义了一个学习使用 MNIST 数据集的临时函数。

第 16、17 行：调用 TensorFlow 的 `input_data` 的 `read_data_sets` 方法，第一个参数为数据集存放路径，第二个参数是标签集的格式。在原始 MNIST 数据集中，我们知道每个样本是 28×28 的黑白图片，对应的是 0~9 的数字标签，所以其格式为：[...784 (28×28) 像素点的值...][3]。其中，第一项为 784 (28×28) 个 0~1 的浮点数，0 代表黑色，1 代表白色；第二项的“3”代表这个样本是数字 3。为了后续处理方便，将标签[3]改为 one-hot 形式，因为标签代表 0~9 的数字，所以标签集为 10 维向量，每维上取值为 0 代表不是这个对应位置的数字，取值为 1 代表是这个对应位置的数字。其中，只有一维可以取 1，因此称之为 one-hot，还以识别结果为数字 3 为例，标签集的格式就变为：[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]，因为第四位为 1，所以代表这个样本是数字 3。

第 18 行：取出训练样本集输入信号集 `X_train`，其为设计矩阵形式，每一行代表一个样本，行数为训练样本集中的样本数量。在这个例子中，就 $X_{\text{train}} \in \mathbb{R}^{55000 \times 784}$ ，训练样本集中有 55000 个样本，每个样本是 784 (28×28) 维的图片。

第 19 行：取出训练样本集标签集 `y_train`，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为训练样本集中的样本数量，在这个例子中，就 $y_{\text{train}} \in \mathbb{R}^{55000 \times 10}$ ，训练样本集中有 55000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 20 行：取出验证样本集输入信号集 `X_validation`，其为设计矩阵形式，每一行代表一个样本，行数为验证样本集中的样本数量。在这个例子中，就 $X_{\text{validation}} \in \mathbb{R}^{5000 \times 784}$ ，验证样本集中有 5000 个样本，每个样本是 784 (28×28) 维的图片。根据前面的讨论可以知道，在训练过程中，为了防止模型出现过拟合，模型的泛化能力降低（模型在训练样本集达到非常高的精度，但是在未见过的测试样本集或实际应用中，精度反而不高），我们通常会采用 Early Stopping 策略，就是在逻辑回归模型训练过程中，只用训练样本集对模型进行训练，每隔一定的时间间隔，计算模型在未见过的验证样本集上识别的精度，并记录迄今为止在验证样本集上取得的最高精度。我们会发现，在训练初期，验证样本集上的识别精度会稳步提高，但是到了一定阶段之后，验证样本集上的识别精度就不再明显提高了，甚至开始逐渐下降，这就说明模型出现了过拟合，这时就可以停止模型训练，将在验证样本集上取得最佳识别精度的模型参数作为模型最终的参数。综上所述，验证样本集主要用于防止模型出现过拟合，为 Early Stopping 算法提供终止依据。

第 21 行：取出验证样本集标签集 `y_validation`，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为训练样本集中的样本数量。在这个例子中，就 $y_{\text{validation}} \in \mathbb{R}^{5000 \times 10}$ ，验证样本集中有 5000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 22 行：取出测试样本集输入信号集 X_{test} ，其为设计矩阵形式，每一行代表一个样本，行数为训练样本集中的样本数量。在这个例子中，就 $X_{\text{test}} \in \mathbb{R}^{10000 \times 784}$ ，训练样本集中有 10000 个样本，每个样本是 784 (28×28) 维的图片。测试样本集主要用于模型训练结束后对模型性能进行评估。由于模型没有见过测试样本集中的样本，可以模拟模型在实际部署之后的情况，模型在测试样本集上的识别精度，基本可以视为模型在实际应用中可以达到的精度。

第 23 行：取出测试样本集标签集 y_{test} ，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为测试样本集中的样本数量。在这个例子中，就 $y_{\text{test}} \in \mathbb{R}^{10000 \times 10}$ ，测试样本集中有 10000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 24~29 行：分别打印训练样本集、验证样本集、测试样本集的内容。

第 30 行：以训练样本集中第 2 个样本为例，带领大家看一下 MNIST 手写数字识别数据集的具体内容。先读出样本输入信号，其为 784 (28×28) 维向量，元素为 0~1 之间的浮点数。由于要进行显示，所以将每个元素乘以 255，变为 0~255 的浮点数，再将其变为 0~255 的整数。

第 31 行：将 784 维向量 `image_raw` 转换为 28×28 的矩阵。

第 32 行：取出该样本的正确结果标签。

第 33 行：定义索引号。

第 34 行：对 one-hot 形式标签循环第 35~37 行的操作。

第 35、36 行：如果标签 one-hot 向量此元素为 1，则终止循环，此时 `idx` 的值就是所对应的数字。

第 37 行：如果标签 one-hot 向量对应此元素位置的值不为 1，则索引号加 1。

第 38 行：将样本标签的正确结果显示在图片标题中。

第 39 行：以灰度图像形式显示样本对应的图片。

第 40 行：具体显示图片。

程序运行结果如图 3.9 和图 3.10 所示。

```
Logistic Regression Startup
Extracting datasets\train-images-idx3-ubyte.gz
Extracting datasets\train-labels-idx1-ubyte.gz
Extracting datasets\t10k-images-idx3-ubyte.gz
Extracting datasets\t10k-labels-idx1-ubyte.gz
X_train: (55000, 784) y_train:(55000, 10)
X_validation: (5000, 784) y_validation:(5000, 10)
X_test: (10000, 784) y_test:(10000, 10)
```

图 3.9 程度运行后台打印结果

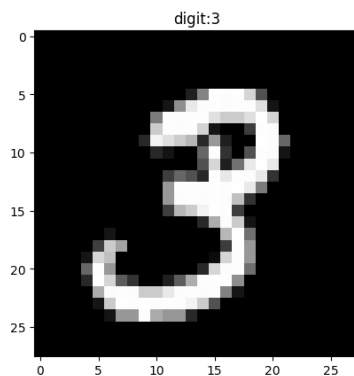


图 3.10 图像及正确结果

下面我们来看逻辑回归引擎类 `Lgr_Engine` 的实现，与上一节的例子不同，在这个例子中将向大家演示如何实现 `Early Stopping`、`L2` 调整项等调整技术，所以需要全新的实现，

代码如下：

```

1 import sys
2 import numpy as np
3 import argparse
4 import matplotlib.pyplot as plt
5 from skimage import io
6 import TensorFlow as tf
7 from TensorFlow.examples.tutorials.mnist import input_data
8
9 class Lgr_Engine(object):
10     # 采用习惯用法定义常量
11     TRAIN_MODE_NEW = 1
12     TRAIN_MODE_CONTINUE = 2
13
14     def __init__(self, datasets_dir):
15         self.datasets_dir = datasets_dir
16         self.batch_size = 100

```

第 11、12 行：定义 train 方法 mode 参数可能的取值：TRAIN_MODE_NEW 代表全新训练模型；TRAIN_MODE_CONTINUE 代表读出以前保存的 ckpt 文件，在此基础上继续进行训练。

第 15 行：定义 datasets_dir 属性，表示 MNIST 数据集存放的位置。

第 16 行：定义 batch_size 属性，表示迷你批次的大小。

下面我们来看读入 MNIST 手写数字识别数据集的方法：

```

1 def load_datasets(self):
2     ''' 调用 TensorFlow 的 input_data，读入 MNIST 手写数字识别数据集的
3     训练样本集、验证样本集、测试样本集
4     '''
5     mnist = input_data.read_data_sets(self.datasets_dir,
6         one_hot=True)
7     X_train = mnist.train.images
8     y_train = mnist.train.labels
9     X_validation = mnist.validation.images
10    y_validation = mnist.validation.labels
11    X_test = mnist.test.images
12    y_test = mnist.test.labels
13    return X_train, y_train, X_validation, y_validation, \
14        X_test, y_test, mnist

```

第 5、6 行：调用 TensorFlow 的 input_data 的 read_data_sets 方法，第一个参数为数据集存放路径，第二个参数是标签集的格式。在原始 MNIST 数据集中，我们知道每个样本是 28×28 的黑白图片，对应的是 0~9 的数字标签，所以其格式为：[...784 (28×28) 像素点的值...][3]。其中，第一项为 784 (28×28) 个 0~1 的浮点数，0 代表黑色，1 代表白色；第二项的“3”代表这个样本是数字 3。为了后续处理方便，我们将标签[3]改为 one-hot 形式，因为标签代表 0~9 的数字，所以标签集为 10 维向量，每维上取值为 0 代表不是这个对应位置的数字，取值为 1 代表是这个对应位置的数字。其中，只有一维可以取 1，因此称之为 one-hot，还以上面的例子为例，标签集的格式就变为：[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]，因为第四位为 1，所以代表这个样本是数字 3。

第7行：取出训练样本集输入信号集 X_{train} ，其为设计矩阵形式，每一行代表一个样本，行数为训练样本集中的样本数量。在这个例子中，就 $X_{\text{train}} \in \mathbb{R}^{55000 \times 784}$ ，其中训练样本集中有 55000 个样本，每个样本是 784 (28×28) 维的图片。

第8行：取出训练样本集标签集 y_{train} ，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为训练样本集中的样本数量。在这个例子中，就 $y_{\text{train}} \in \mathbb{R}^{55000 \times 10}$ ，其中训练样本集中有 55000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第9行：取出验证样本集输入信号集 $X_{\text{validation}}$ ，其为设计矩阵形式，每一行代表一个样本，行数为验证样本集中的样本数量。在这个例子中，就 $X_{\text{validation}} \in \mathbb{R}^{5000 \times 784}$ ，其中验证样本集中有 5000 个样本，每个样本是 784 (28×28) 维的图片。根据前面我们的讨论可以知道，在训练过程中，为了防止模型出现过拟合，模型的泛化能力降低（模型在训练样本集达到非常高的精度，但是在未见过的测试样本集或实际应用中，精度反而不高），通常会采用 Early Stopping 策略，就是在逻辑回归模型训练过程中，只用训练样本集对模型进行训练，每隔一定的时间间隔，计算模型在未见过的验证样本集上识别的精度，并记录迄今为止在验证样本集上取得的最高精度。我们会发现，在训练初期，验证样本集上的识别精度会稳步提高，但是到了一定阶段之后，验证样本集上的识别精度就不会再明显提高了，甚至开始逐渐下降，这就说明模型出现了过拟合，这时就可以停止模型训练，将在验证样本集上取得最佳识别精度的模型参数作为模型最终的参数。综上所述，验证样本集主要用于防止模型出现过拟合，为 Early Stopping 算法提供终止依据。

第10行：取出验证样本集标签集 $y_{\text{validation}}$ ，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为训练样本集中的样本数量。在这个例子中，就 $y_{\text{validation}} \in \mathbb{R}^{5000 \times 10}$ ，其中验证样本集中有 5000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第11行：取出测试样本集输入信号集 X_{test} ，其为设计矩阵形式，每一行代表一个样本，行数为训练样本集中的样本数量。在这个例子中，就 $X_{\text{test}} \in \mathbb{R}^{10000 \times 784}$ ，其中训练样本集中有 10000 个样本，每个样本是 784 (28×28) 维的图片。测试样本集主要用于模型训练结束后对模型性能进行评估。由于模型没有见过测试样本集中的样本，可以模拟模型在实际部署之后的情况，模型在测试样本集上的识别精度，基本可以视为模型在实际应用中可以达到的精度。

第12行：取出测试样本集标签集 y_{test} ，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为测试样本集中的样本数量。在这个例子中，就 $y_{\text{test}} \in \mathbb{R}^{10000 \times 10}$ ，其中测试样本集中有 10000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第13、14行：返回训练样本集、验证样本集、测试样本集输入信号集和标签集。

下面来看怎样建立一个在实际中应用的完整逻辑回归模型，代码如下：

```
1 def build_model(self):
2     X = tf.placeholder(tf.float32, [None, 784])
3     W = tf.Variable(tf.zeros([784, 10]))
```



```

4  b = tf.Variable(tf.zeros([10]))
5  #y_ = tf.matmul(X, W) + b
6  z = tf.matmul(X, W) + b
7  y_ = tf.nn.softmax(z)
8  y = tf.placeholder(tf.float32, [None, 10])
9  cross_entropy = tf.reduce_mean(
10     tf.nn.softmax_cross_entropy_with_logits(labels=y, logits=z))
11  lanmeda = 0.001
12  J = cross_entropy + lanmeda*tf.reduce_sum(W**2)
13  #train_step = tf.train.GradientDescentOptimizer(0.5).minimize(
14  #    cross_entropy)
15  train_step = tf.train.GradientDescentOptimizer(0.5).minimize(J)
16  correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
17  accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
18  return X, W, b, y_, y, cross_entropy, train_step, \
19     correct_prediction, accuracy

```

第 2 行：定义存放输入信号集的 `placeholder` 类型的 `X`，每一行代表一个样本，维度为 784（28×28）维，我们一次会处理一个迷你批次，因此 `X` 的第一维为 `None`，在程序实际运行时，会根据迷你批次的大小给第一维赋上实际的值。

第 3 行：定义输入层和输出层的连接权值矩阵，输入层为 784（28×28），输出层为 0~9 的 10 个类别，数量为 10，因此为 784×10 维的连接权值矩阵。

第 4 行：定义输出层神经元的偏移量向量，维度为 10。

第 6 行：定义每个输出层神经元接收到的输入层输出经过仿射变换后的线性和，变为下标形式后如下：

$$z_i = \sum_{j=1}^{784} W_{i,j} x_i, i = 1, 2, \dots, 10$$

第 7 行：利用 `softmax` 函数，将输入信号 `z` 变为每个类别的概率，公式为：

$$y_{-i} = \frac{e^{z_i}}{\sum_{j=1}^{10} e^{z_j}}$$

上式表明每个类别的概率，所有类别的概率值之和为 1，我们认为概率最大的类别就是识别结果。注意：其实我们为了计算方便，这里直接计算了 0~9 个数字对应的 10 个类别，根据 10 个类别的概率之和为 1，我们只需要计算 9 个类别就可以了，第 10 个类别可以用 1 减去其余 9 个类别得到。但是为了方便计算，我们通常都计算 10 个类别，即使这样会增加一些运算量。

第 8 行：定义正确识别结果 `y` 对应的 `placeholder`。

第 9、10 行：定义交叉熵 `cross_entropy` 的计算方法。

第 11 行：定义调整项 L2 权值衰减系数 λ 。

第 12 行：定义最终代价函数 `J`，值为交叉熵再加上 L2 调整项（权值衰减）。

第 15 行：采用梯度下降算法，学习系数为 0.5，利用优化算法求使代价函数 `J` 在到最小值时连接权值 `W` 和偏移量 `b` 的值。

第 16 行：利用 TensorFlow 的 `argmax` 函数，分别求出我们的计算类别向量每个样本的最大值下标和类别向量每个样本的最大值下标，并对其进行比较。

第 17 行：求出预测精度，首先调用 TensorFlow 的 `cast` 函数，将第 16 行的结果变为浮点数列表：`[1.0, 1.0, 0.0, 1.0, 1.0]`，这里假设只有 5 个样本。再调用 TensorFlow 的 `reduce_mean` 函数求出这个列表的平均值： $(1.0+1.0+0.0+1.0+1.0)/5=0.8$ ，这个值就是模型预测的精度。

第 18、19 行：返回模型定义的 `X, W, b, y_, y, cross_entropy, train_step, correct_prediction, accuracy`。

下面我们来看最重要的训练方法，在这个方法中，将向大家演示为了防止出现过拟合而采用的 `Early Stopping` 技术，代码如下：

```

1 def train(self, mode=TRAIN_MODE_NEW, ckpt_file='work/lgr.ckpt'):
2     X_train, y_train, X_validation, y_validation, X_test, \
3         y_test, mnist = self.load_datasets()
4     X, W, b, y_, y, cross_entropy, train_step, correct_prediction, \
5         accuracy = self.build_model()
6     epochs = 10
7     saver = tf.train.Saver()
8     total_batch = int(mnist.train.num_examples/self.batch_size)
9     check_interval = 50
10    best_accuracy = -0.01
11    improve_threthold = 1.005
12    no_improve_steps = 0
13    max_no_improve_steps = 3000
14    is_early_stop = False
15    with tf.Session() as sess:
16        sess.run(tf.global_variables_initializer())
17        if Lgr_Engine.TRAIN_MODE_CONTINUE == mode:
18            saver.restore(sess, ckpt_file)
19        for epoch in range(epochs):
20            if is_early_stop:
21                break
22            for batch_idx in range(total_batch):
23                if no_improve_steps >= max_no_improve_steps:
24                    is_early_stop = True
25                    break
26                X_mb, y_mb = mnist.train.next_batch(self.batch_size)
27                sess.run(train_step, feed_dict={X: X_mb, y: y_mb})
28                no_improve_steps += 1
29                if batch_idx % check_interval == 0:
30                    train_accuracy = sess.run(accuracy,
31                        feed_dict={X: X_train, y: y_train})
32                    validation_accuracy = sess.run(accuracy,
33                        feed_dict={X: X_validation, y: y_validation})
34                    if best_accuracy < validation_accuracy:
35                        if validation_accuracy / best_accuracy >= \
36                            improve_threthold:
37                            no_improve_steps = 0
38                            best_accuracy = validation_accuracy
39                            saver.save(sess, ckpt_file)
40                    print('{0}:{1}# train:{2}, validation:{3}'.format(
41                        epoch, batch_idx, train_accuracy,
42                        validation_accuracy))

```

```

43     print(sess.run(accuracy, feed_dict={X: X_test,
44                                         y: y_test}))
45

```

第 2、3 行：读入训练样本集输入信号集、训练样本集标签集、验证样本集输入信号集、验证样本集标签集、测试样本集输入信号集、测试样本集标签集。

第 4、5 行：创建模型。

第 6 行：学习整个训练样本集遍数。

第 7 行：初始化 TensorFlow 模型保存和恢复对象 saver。

第 8 行：用用户训练样本集中的样本数除以迷你批次大小，得到迷你批次数量 total_batch。

第 9 行：每隔 50 次迷你批次学习，计算在验证样本集上的精度。

第 10 行：保存在验证样本集上所取得的最好的验证样本集的精度。

第 11 行：定义验证样本集上精度提高 0.5% 时才算显著提高。

第 12 行：记录在验证样本集上精度没有显著提高学习迷你批次的次数。

第 13 行：在验证样本集精度最大没有显著提高的情况下，允许学习迷你批次的次数。

第 14 行：是否终止学习过程。

第 15 行：启动 TensorFlow 会话。

第 16 行：初始化全局参数。

第 17、18 行：如果模式为 TRAIN_MODE_CONTINUE，则读入以前保存的 ckpt 模型文件，初始化模型参数。

第 19 行：循环第 20~42 行操作，对整个训练样本集进行一次学习。

第 20、21 行：如果 is_early_stop 为真，则终止本层循环。

第 22 行：循环第 23~42 行操作，对一个迷你批次进行学习。

第 23、25 行：如果验证样本集上识别精度没有显著改善的迷你批次学习次数大于最大允许的验证样本集上识别精度没有显著改善的迷你批次学习次数，则将 is_early_stop 置为真，并退出本层循环。这会直接触发第 20、21 行终止外层循环，学习过程结束。

第 26 行：从训练样本集中取出一个迷你批次的输入信号集 X_mb 和标签集 y_mb。

第 27 行：调用 TensorFlow 计算模型输出、代价函数，求出代价函数对参数的导数，并应用梯度下降算法更新模型参数值。

第 28 行：将验证样本集没有显著改善的迷你批次学习次数加 1。

第 29 行：如果连续进行了指定次数的迷你批次学习，则计算统计信息。

第 30、31 行：计算训练样本集上的识别精度。

第 32、33 行：计算验证样本集上的识别精度。

第 34 行：如果验证样本集上最佳识别精度小于当前验证样本集上的识别精度，执行第 35~39 行操作。

第 35~37 行：如果当前验证样本集上的识别精度比之前的最佳识别精度提高 0.5% 以上，则将验证样本集没有显著改善的迷你批次学习次数设为 0。

第 38 行：将验证样本集上最佳识别精度的值设置为当前验证样本集上的识别精度值。

第 39 行：将当前模型参数保存到 ckpt 模型文件中。

第 40~42 行：打印训练状态信息。

第 43、44 行：训练完成后，计算测试样本集上的识别精度，并打印出来。

运行程序，得到如图 3.11（片段）所示结果。

```
0:0# train:0.3383636474609375, validation:0.34299999475479126
0:50# train:0.7101272940635681, validation:0.7138000130653381
0:100# train:0.7923091053962708, validation:0.8051999807357788
0:150# train:0.8041090965270996, validation:0.8123999834060669
0:200# train:0.8086363673210144, validation:0.8191999793052673
0:250# train:0.8134363889694214, validation:0.8209999799728394
0:300# train:0.8184727430343628, validation:0.8289999961853027
0:350# train:0.8511636257171631, validation:0.8623999953269958
0:400# train:0.8712727427482605, validation:0.8809999823570251
0:450# train:0.8792726993560791, validation:0.8889999985694885
0:500# train:0.8839091062545776, validation:0.8913999795913696

5:300# train:0.9008181691169739, validation:0.9082000255584717
5:350# train:0.9008727073669434, validation:0.907800018787384
5:400# train:0.9009090662002563, validation:0.9101999998092651
5:450# train:0.9009818434715271, validation:0.9065999984741211
5:500# train:0.9008908867835999, validation:0.9064000248908997
6:0# train:0.9012908935546875, validation:0.9083999991416931
6:50# train:0.9014909267425537, validation:0.9065999984741211
6:100# train:0.9002545475959778, validation:0.9085999727249146
6:150# train:0.9015091061592102, validation:0.9085999727249146
0.9099
```

图 3.11 逻辑回归模型训练过程

最终在测试样本集上可以达到的精度为 90.99%，而在训练样本集上的精度为 90.15%，验证样本集上的精度为 0.90.88%。这个结果说明：首先我们采用 L2 调整项（权值衰减），而且采用 Early Stopping 算法，这些调整项都是降低训练样本集的精度，提高测试样本集的精度，如果运用得当，可以提高模型的泛化能力（对新样本的正确识别能力）。我们只给出了这些调整项，但是对于权值衰减系数、验证样本集上识别精度显著提高标准、允许验证样本集识别精度不提高的最大次数等超参数，并没有仔细地进行调优，所以并不是最佳值。如果去掉这些调整项，测试样本集的识别精度可以达到 92% 左右，读者可以尝试改变与这些调整项相关的超参数，看看能不能取得更好的识别精度。但是需要注意的是，MNIST 手写字母识别数据集，从本质上来讲，是线性不可分的，因此像逻辑回归这种线性模型，能够达到的最佳精度的提升空间是非常有限的。

模型训练完成之后，我们就可以用模型来对未知样本进行识别了，run 方法的代码如下：

```
1 def run(self, ckpt_file='work/lgr.ckpt'):
2     img_file = 'datasets/test6.png'
3     img = io.imread(img_file, as_grey=True)
4     raw = [1 if x<0.5 else 0 for x in img.reshape(784)]
5     #sample = np.array(raw)
6     X_train, y_train, X_validation, y_validation, \
7         X_test, y_test, mnist = self.load_datasets()
8     X, W, b, y_, y, cross_entropy, train_step, correct_prediction, \
9         accuracy = self.build_model()
10    sample = X_test[102]
11    img_in = sample.reshape(28, 28)
12    X_run = sample.reshape(1, 784)
13    saver = tf.train.Saver()
14    digit = -1
```

```

15     with tf.Session() as sess:
16         sess.run(tf.global_variables_initializer())
17         saver.restore(sess, ckpt_file)
18         rst = sess.run(y_, feed_dict={X: X_run})
19         print('rst:{0}'.format(rst))
20         max_prob = -0.1
21         for idx in range(10):
22             if max_prob < rst[0][idx]:
23                 max_prob = rst[0][idx]
24                 digit = idx;
25     plt.imshow(img_in, cmap='gray')
26     plt.title('result:{0}'.format(digit))
27     plt.axis('off')
28     plt.show()

```

第 2 行：用绘图软件做一个 28×28 的图像，在上面写一个数字，这里我们直接写上一个印刷体的 5。

第 3 行：以灰度图像方式读出图像内容。

第 4 行：首先将其形状从二维 28×28 变为一维 784，然后根据每个像素的值进行处理：值小于 0.5 时取 1，否则取 0。

第 5 行：将其变为 numpy 数组，作为一个样本。

第 6、7 行：调用 load_datasets 方法，读入训练样本集、验证样本集、测试样本集的内容。

第 8、9 行：调用 build_model 方法，建立逻辑回归模型。

第 10 行：取测试样本集中的第 103 个样本（下标 0 为第 1 个样本）作为测试样本，由于我们的模型在训练过程中没有见过测试样本集中的样本，因此可以模拟实际应用中遇到的情况。

第 11 行：将这个样本的形状变为 28×28，方便进行图像显示。

第 12 行：将样本从一维 784 变为二维 1×784，作为模型的输入样本 X_run。

第 13 行：初始化 TensorFlow 的 saver 对象。

第 14 行：定义 digit 为最终识别出的 0~9 的数字，取 -1 表示还没有识别结果。

第 15 行：启动 TensorFlow 会话来运行程序。

第 16 行：初始化变量。

第 17 行：恢复之前保存的模型参数文件，并初始化模型参数。

第 18 行：求以样本 X_run 为输入，在输出层经过 softmax 函数后的计算值，表示为每个类别的概率。

第 19 行：打印出所有类别的概率值。

第 20 行：定义 max_prob 记录所有类别最大的概率值。

第 21 行：循环识别结果 rst 每个类别的概率。

第 22~24 行：如果最大概率小于当前类别的概率，将当前概率赋给最大概率，识别出的数字等于类别索引值，即对应的 0~9 中的数字。当循环完所有类别后，我们就能找到概率最大的类别及其对应的数字了。

第 25 行：以黑白图像方式，利用 matplotlib 来显示图像。

第 26 行：将识别结果作为图像的标题进行显示。

第 27 行：不显示横、纵坐标轴

第 28 行：具体显示要检测的图像。

我们要识别的图片如图 3.12 所示。

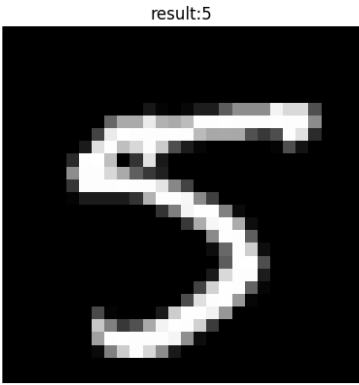


图 3.12 要识别的图片

这么模糊的图像逻辑回归模型都能识别出正确的结果，看来它的功能很强大。

运行上面的程序得到如图 3.13 所示结果。

```
rst:[[ 1.27761872e-04  5.90814636e-07  2.59901071e-07  7.50286097e-04
 3.92440052e-05  9.91870761e-01  7.78191588e-06  1.69550523e-03
 4.67756437e-03  8.30283971e-04]]
```

图 3.13 模型运行结果输出

由上图可以看出，下标为 5 处概率最大，识别结果应该是数字 5，与正确的类别一致，表明逻辑回归模型训练是成功的。

下面再来看一下利用自己的图片文件，让逻辑回归模型进行识别会产生什么结果。生成的图片如图 3.14 所示。



图 3.14 人工生成数字图片

运行上面的程序，得到如图 3.15 和图 3.16 所示结果。

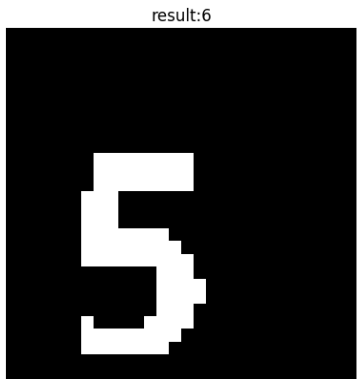


图 3.15 人工生成图片识别结果

```
rst:[[ 1.94366947e-02  5.96820726e-04  1.29055970e-05  1.05798279e-06
 4.46188264e-02  3.24702077e-02  8.03695202e-01  2.46558059e-03
 8.86678249e-02  8.03481694e-03]]
```

图 3.16 人工生成图片后台输出

由上图可以看出，用印刷体生成的图片文件，我们识别起来还是比较简单的，但是逻辑回归模型却会给出错误的分类结果，这就说明模型还是有一定的缺陷的。在后面的章节中，我们将向大家介绍怎样避免这种情况发生。出现这种情况的根本原因是假设逻辑回归模型拟合的函数具有局部光滑性，其可以根据附近样本来预测当前样本的类别。当样本出现一定的扰动后，就有可能落到其他类别的附近区域了，因此会出现误判，这也是对抗样本训练方法提出的背景。对抗样本就是人工生成一些我们看不出区别，但是模型却会给出完全错误的分类结果的样本，通过对这些对抗样本的学习，可以极大地改善模型的泛化能力。在这方面还没有非常好的解决方案，读者可以尝试做一些类似上面的样本，加入到训练样本集中，对逻辑回归模型进行训练，看看是不是可以提高模型的泛化能力。

到目前为止，我们详细介绍了逻辑回归算法及其应用。与网络上常见的教程不同，我们在这里给出的例程，都采用 L2 调整项（权值衰减）和 Early Stopping 技术。为了解决模型的过拟合问题，提高模型的泛化能力，这些技术在实际项目中被广泛使用。读者可以在此基础上，在实际项目中构建自己的逻辑回归模型。

第 4 章

感知器模型和 MLP

本章将向读者介绍前馈神经网络，即在神经网络中没有回路，只能前向传播。实际上，还有一类神经网络，可以构成环状结构，被称为反馈神经网络或递归神经网络，这种网络的结构比较复杂，但是表现能力更强，在语音识别等时序分析领域应用非常广泛，我们将在后面的章节进行介绍。本章介绍的前馈神经网络，也被称为多层感知器（MLP）模型，是最基础的神经网络之一。

本章首先介绍神经元的基本模型、感知器模型及多层感知器模型的学习算法，即 BP 算法。然后介绍 BP 算法的 TensorFlow 实现。最后将以 MNIST 手写数字识别数据集为例，讲解怎样采用多层感知器模型来实现模式分类功能。

4.1 感知器模型

4.1.1 神经元模型

神经网络是由神经元组成的，神经元模型如图 4.1 所示，左侧方框代表输入信号，假设输入信号为 $\mathbf{x} \in \mathbb{R}^n$ ，是 n 维向量形式，通常为了提高运算效率，会引入额外节点 $x_0 = 1$ 。输入信号经过输入突触进入神经元，每个输入突触均有一个权值，表示输入向量 \mathbf{x} 在对应位置分量的权重，用 w_i 来表示。同时规定， w_0 为神经元的偏置量。

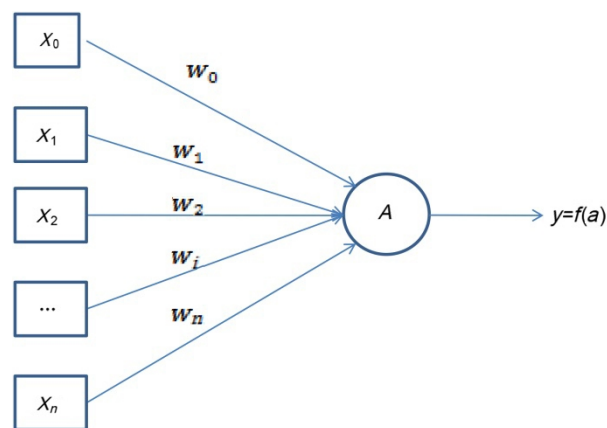


图 4.1 神经元模型

有了上面的定义之后，神经元的输入可以表示为：

$$a = w_0x_0 + w_1x_1 + \cdots + w_nx_n = \sum_{i=0}^n w_ix_i = \mathbf{w}^T \mathbf{x}$$

式中， $\mathbf{w} \in \mathbb{R}^{n+1}$ 、 $\mathbf{x} \in \mathbb{R}^{n+1}$ ，均为向量形式。

神经元的输出可以表示为：

$$y = f(a)$$

式中， f 为神经元的激活函数，根据神经元所处的位置不同，有多种形式可供选择。下面我们就来介绍其中最常用的四种形式。

1. Sigmoid 神经元

神经元的激活函数采用 Sigmoid 函数，这是神经网络早期发展过程中最常用的一种激活函数，其与生物神经元比较类似。当前，如果我们的分类问题有两类，则通常在输出层神经元上采用这种激活函数。

Sigmoid 函数定义如下：

$$g(z) = \frac{1}{1 + e^{-z}}$$

其输出值范围为(0,1)，可以解释为类别出现的概率。其函数图形如图 4.2 所示 (chp04/c001.py)。

由图 4.2 可以看出，当 z 值过大或过小时，函数值将接近 0 或 1，所以存在饱和现象，这也是这种神经元在前馈网络隐藏层不再流行的原因。所以在实际应用中，我们需要尽量使输入信号位于(-5.0, 5.0)。

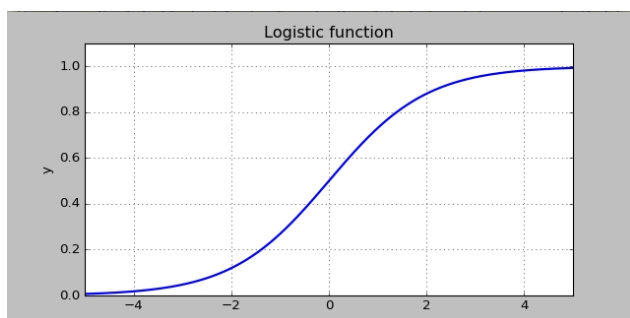


图 4.2 Sigmoid 函数图像

对于 $g(z)$ 这个 Sigmoid 函数，在深度学习算法中会经常用到，而且通常会用到其导数，下面进行推导：

$$\begin{aligned}
 g'(z) &= \frac{d}{dz} \frac{1}{1 + e^{-z}} = \left(-\frac{1}{(1 + e^{-z})^2} \right) (-e^{-z}) \\
 &= \frac{e^{-z}}{(1 + e^{-z})^2} = \frac{1}{1 + e^{-z}} \left(\frac{e^{-z}}{1 + e^{-z}} \right) \\
 &= \frac{1}{1 + e^{-z}} \left(1 - \frac{1}{1 + e^{-z}} \right) \\
 &= g(z)(1 - g(z))
 \end{aligned}$$

2. 双曲正切神经元

人们发现，如果将 Sigmoid 函数作为隐藏层神经元的激活函数，那么神经网络的学习速度将变慢，而双曲正切函数在性能方面会优于 Sigmoid 函数，因此在前馈神经网络隐藏层中，通常采用双曲正切函数。

双曲正切函数的定义如下：

$$g(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

其函数图像如图 4.3 所示（chp04/c002.py）。

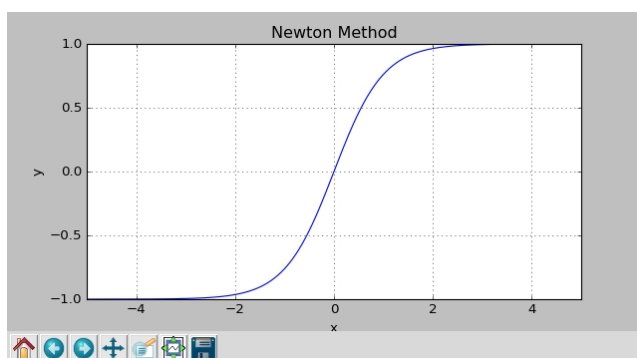


图 4.3 双曲正切函数图像

由图 4.3 可以看出，双曲正切函数的值域为 $(-1, 1)$ ，当 x 小于 -2 或大于 2 时，函数值会非常接近 -1 或 1 ，出现饱和现象，因此输入值最好位于 $(-2, 2)$ 。

在神经网络中，经常会用到双曲正切函数的导数，下面推导双曲正切函数的求导公式：

$$\begin{aligned} g'(x) &= \tanh'(x) = \left(\frac{e^x - e^{-x}}{e^x + e^{-x}} \right)' \\ &= \left((e^x - e^{-x}) \frac{1}{e^x + e^{-x}} \right)' \quad \# \text{ 乘积导数公式: } (AB)' = A'B + AB' \\ &= (e^x - e^{-x})' \frac{1}{e^x + e^{-x}} + (e^x - e^{-x}) \left(\frac{1}{e^x + e^{-x}} \right)' \\ &= (e^x + e^{-x}) \frac{1}{e^x + e^{-x}} - (e^x - e^{-x}) \frac{1}{(e^x + e^{-x})^2} (e^x - e^{-x}) \\ &= 1 - \left(\frac{e^x - e^{-x}}{e^x + e^{-x}} \right)^2 \\ &= 1 - (\tanh(x))^2 \end{aligned}$$

其实 Sigmoid 函数与双曲正切函数存在 $\tanh(x) = 2 \times \text{sigmoid}(2x) - 1$ 的关系，因此两个函数在很大程度上是等价的，只是当双曲正切函数用于隐藏层时，收敛速度会更快一些。而且，双曲正切函数一般不用于输出层。

3. 改进线性单元

改进线性单元（ReLU）函数是近些年发展起来的一种激活函数，与 Sigmoid 函数相比，其不具有生物神经元的相似性，最初并不被人们所重视，但是由于其具有简单的特点，同时没有 Sigmoid 函数等在输入信号过大或过小时出现的饱和现象，所以它逐渐成为前馈神经网络隐藏层主流的神经元模型。

改进线性单元（ReLU）函数的定义为：

$$g(x) = \max\{0, x\}$$

其函数图像如图 4.4 所示（chp04/c003.py）。

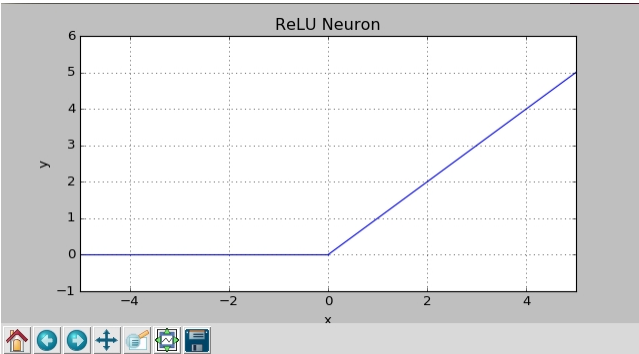


图 4.4 ReLU 函数图像

当神经元的输入值为负数时，神经元的输出值为 0，对其求导为 0。当神经元的输入值为正数时，神经元的输出值即为输入值，此时对其求导为 1。除 $x=0$ 点外，ReLU 函数的其他各点均可导。实践证明，使用 ReLU 函数作为隐藏层神经元的激活函数，神经网络的性能更好，其与将代价函数从最小平方误差变为负对数似然函数一起，成为近 10 年来前馈神经网络最重要的进展之一。

由于 ReLU 函数在输入值为负数时导数为 0，因此无法通过梯度下降算法来调整连接权值，所以人们发明了各种 ReLU 的泛化模型，这些模型基本上性能相当，其中比较有代表性的是下面这种：

$$g(x_i|\alpha_i) = \max(0, x_i) + \alpha_i \min(0, x_i)$$

当 $\alpha_i = -1$ 时，其就变为绝对值公式，即 $g(x)=|x|$ ，这种形式在图像物体识别中得到了广泛应用。

4. softmax 神经元

这种神经元主要用于神经网络的输出层，当模式分类类别数 $K \geq 3$ 时使用。其具体定义如下：

$$y_i = \frac{e^{a_i}}{\sum_{k=1}^K e^{a_k}}$$

式中， a_i 为神经元的输入值。在本章进行 MNIST 手写数字识别的实例中，模式分类的类别数为 10，因此采用这种神经元。

4.1.2 神经网络架构

本节将以如图 4.5 所示的网络结构为例来讲解多层感知器模型。

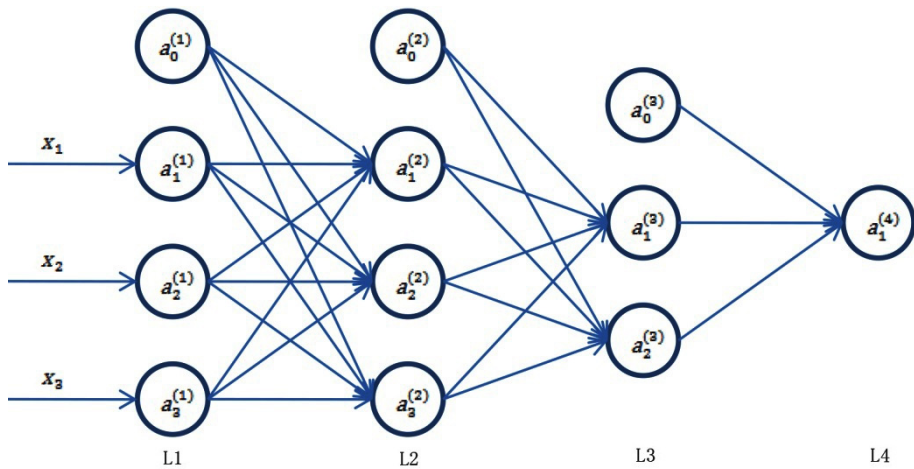


图 4.5 多层感知器网络架构

在图 4.5 中，输入向量 $\mathbf{x} \in \mathbb{R}^n$ ， n 为输入向量维度，即模型的特征数，分量为 $x_i (i \in \{1, 2, \dots, n\})$ 。对图 4.5 中每个神经元的符号描述如下。

$a_i^{(j)}$ 的上标 j 代表在第 j 层，下标 i 代表该层第 i 个节点，且 $i > 0$ 时代表第 j 层的第 i 个神经元；当 $i=0$ 时，是我们为了将神经网络中的计算转变为向量计算而引入的辅助节点；当 $a_0^{(j)} = 1$ 时，其向下一层的连接权值分别对应下一层神经元偏置值（Bias）。其中，第 j 层到第 $j+1$ 层的连接权值矩阵为 $\mathbf{W}^{(j)}$ ，其分量 $W_{ih}^{(j)}$ 表示第 j 层第 h 个节点到第 $j+1$ 层第 i 个节点的连接权值。

令每个训练样本的格式为 $(\mathbf{x}^{(s)}, \mathbf{y}^{(s)})$ ，表示第 s 个训练样本。

4.2 数值计算形式

4.2.1 前向传播

有了上述符号之后，就可以开始进行前向传播计算了。

程序刚开始运行时，假设运行的是第 s 个样本集，第 1 层可以得到：

$$\begin{aligned} a_0^{(1)} &= 1, \\ a_1^{(1)} &= x_1^{(s)}, \\ a_2^{(1)} &= x_2^{(s)}, \\ &\dots \\ a_n^{(1)} &= x_n^{(s)} \end{aligned}$$

第 1 层输出值计算出来之后，就可以开始计算第 2 层的输入值和输出值，假设输入值用 z 表示，则输入值为：

$$\begin{aligned} z_1^{(2)} &= W_{10}^{(1)} a_0^{(1)} + W_{11}^{(1)} a_1^{(1)} + W_{12}^{(1)} a_2^{(1)} + W_{13}^{(1)} a_3^{(1)}, \\ z_2^{(2)} &= W_{20}^{(1)} a_0^{(1)} + W_{21}^{(1)} a_1^{(1)} + W_{22}^{(1)} a_2^{(1)} + W_{23}^{(1)} a_3^{(1)}, \\ z_3^{(2)} &= W_{30}^{(1)} a_0^{(1)} + W_{31}^{(1)} a_1^{(1)} + W_{32}^{(1)} a_2^{(1)} + W_{33}^{(1)} a_3^{(1)} \end{aligned}$$

计算出输入值之后，就可计算输出值，假设激活函数为 f_h ，则有如下公式：

$$\begin{aligned} a_0^{(2)} &= 1, \\ a_1^{(2)} &= f_h(z_1^{(2)}), \end{aligned}$$

$$a_2^{(2)} = f_h(z_2^{(2)}),$$

$$a_3^{(2)} = f_h(z_3^{(2)})$$

接下来计算第 3 层的输入值：

$$z_1^{(3)} = W_{10}^{(2)} a_0^{(2)} + W_{11}^{(2)} a_1^{(2)} + W_{12}^{(2)} a_2^{(2)} + W_{13}^{(2)} a_3^{(2)},$$

$$z_2^{(3)} = W_{20}^{(2)} a_0^{(2)} + W_{21}^{(2)} a_1^{(2)} + W_{22}^{(2)} a_2^{(2)} + W_{23}^{(2)} a_3^{(2)},$$

下面计算第 3 层的输出值：

$$a_0^{(3)} = 0,$$

$$a_1^{(3)} = f_h(z_1^{(3)}),$$

$$a_2^{(3)} = f_h(z_2^{(3)})$$

再下面就是输出层，根据我们的讨论，输出层通常与隐藏层有不同的激活函数，在这里设其激活函数为 f_o ，第 4 层输入值为：

$$z_1^{(4)} = W_{10}^{(3)} a_0^{(3)} + W_{11}^{(3)} a_1^{(3)} + W_{12}^{(3)} a_2^{(3)}$$

输出值为：

$$a_1^{(4)} = f_o(z_1^{(4)})$$

至此，单个训练样本 $(\mathbf{x}^{(s)}, \mathbf{y}^{(s)})$ 的前向传播就完成了。下面来看看误差反向传播部分。

4.2.2 误差反向传播

在求出输出层输出之后（在本例中为 $a_1^{(4)}$ ），我们就可以先将其与训练样本的期望输出值进行比较，求出误差，然后按照与输入信号前向传播类似的方法进行误差反向传播。

下面计算第 4 层，也就是输出层误差：

$$\delta_1^{(4)} = y_1^{(s)} - a_1^{(4)}$$

将误差从第 4 层传到第 3 层：

$$\delta_1^{(3)} = f_h'(z_1^{(3)}) (W_{11}^{(3)} \delta_1^{(4)}),$$

$$\delta_2^{(3)} = f_h'(z_2^{(3)}) (W_{12}^{(3)} \delta_1^{(4)})$$

调整第 3 层到第 4 层的连接权值，假设本次为第 n 次调整，首先计算权值调整量，然后给出权值调整公式：

$$\Delta W_{10}^{(3)(t)} = -\alpha \delta_1^{(4)} a_0^{(3)},$$

$$\Delta W_{11}^{(3)(t)} = -\alpha \delta_1^{(4)} a_1^{(3)},$$

$$\Delta W_{12}^{(3)(t)} = -\alpha \delta_1^{(4)} a_2^{(3)}$$

式中， α 为学习率， t 为第 t 次进行调整。则权值调整公式为：

$$W_{10}^{(3)(t)} = W_{10}^{(3)(t-1)} + \Delta W_{10}^{(3)(t)} + m\Delta W_{10}^{(3)(t-1)} = W_{10}^{(3)(t-1)} - \alpha \delta_1^{(4)} a_0^{(3)} + m\Delta W_{10}^{(3)(t-1)},$$

$$W_{11}^{(3)(t)} = W_{11}^{(3)(t-1)} + \Delta W_{11}^{(3)(t)} + m\Delta W_{11}^{(3)(t-1)} = W_{11}^{(3)(t-1)} - \alpha \delta_1^{(4)} a_1^{(3)} + m\Delta W_{11}^{(3)(t-1)},$$

$$W_{12}^{(3)(t)} = W_{12}^{(3)(t-1)} + \Delta W_{12}^{(3)(t)} + m\Delta W_{12}^{(3)(t-1)} = W_{12}^{(3)(t-1)} - \alpha \delta_1^{(4)} a_2^{(3)} + m\Delta W_{12}^{(3)(t-1)}$$

式中， m 为动量项，可以加速算法收敛过程，也可以避免算法陷入局部最小值点， $\Delta W_{11}^{(3)(t)}$ （等式右侧的项）为上一次权值调整的值。

将误差从第3层传到第2层，先计算第 t 次调整量：

$$\Delta W_{10}^{(2)(t)} = -\alpha \delta_1^{(3)} a_0^{(2)},$$

$$\Delta W_{11}^{(2)(t)} = -\alpha \delta_1^{(3)} a_1^{(2)},$$

$$\Delta W_{12}^{(2)(t)} = -\alpha \delta_1^{(3)} a_2^{(2)},$$

$$\Delta W_{13}^{(2)(t)} = -\alpha \delta_1^{(3)} a_3^{(2)},$$

$$\Delta W_{20}^{(2)(t)} = -\alpha \delta_2^{(3)} a_0^{(2)},$$

$$\Delta W_{21}^{(2)(t)} = -\alpha \delta_2^{(3)} a_1^{(2)},$$

$$\Delta W_{22}^{(2)(t)} = -\alpha \delta_2^{(3)} a_2^{(2)},$$

$$\Delta W_{23}^{(2)(t)} = -\alpha \delta_2^{(3)} a_3^{(2)}$$

权值调整公式为：

$$W_{10}^{(2)(t)} = W_{10}^{(2)(t-1)} + \Delta W_{10}^{(2)(t)} + m\Delta W_{10}^{(2)(t-1)} = W_{10}^{(2)(t-1)} - \alpha \delta_1^{(3)} a_0^{(2)} + m\Delta W_{10}^{(2)(t-1)},$$

$$W_{11}^{(2)(t)} = W_{11}^{(2)(t-1)} + \Delta W_{11}^{(2)(t)} + m\Delta W_{11}^{(2)(t-1)} = W_{11}^{(2)(t-1)} - \alpha \delta_1^{(3)} a_1^{(2)} + m\Delta W_{11}^{(2)(t-1)},$$

$$W_{12}^{(2)(t)} = W_{12}^{(2)(t-1)} + \Delta W_{12}^{(2)(t)} + m\Delta W_{12}^{(2)(t-1)} = W_{12}^{(2)(t-1)} - \alpha \delta_1^{(3)} a_2^{(2)} + m\Delta W_{12}^{(2)(t-1)},$$

$$W_{13}^{(2)(t)} = W_{13}^{(2)(t-1)} + \Delta W_{13}^{(2)(t)} + m\Delta W_{13}^{(2)(t-1)} = W_{13}^{(2)(t-1)} - \alpha \delta_1^{(3)} a_3^{(2)} + m\Delta W_{13}^{(2)(t-1)},$$

$$W_{20}^{(2)(t)} = W_{20}^{(2)(t-1)} + \Delta W_{20}^{(2)(t)} + m\Delta W_{20}^{(2)(t-1)} = W_{20}^{(2)(t-1)} - \alpha \delta_2^{(3)} a_0^{(2)} + m\Delta W_{20}^{(2)(t-1)},$$

$$W_{21}^{(2)(t)} = W_{21}^{(2)(t-1)} + \Delta W_{21}^{(2)(t)} + m\Delta W_{21}^{(2)(t-1)} = W_{21}^{(2)(t-1)} - \alpha \delta_2^{(3)} a_1^{(2)} + m\Delta W_{21}^{(2)(t-1)},$$

$$W_{22}^{(2)(t)} = W_{22}^{(2)(t-1)} + \Delta W_{22}^{(2)(t)} + m\Delta W_{22}^{(2)(t-1)} = W_{22}^{(2)(t-1)} - \alpha \delta_2^{(3)} a_2^{(2)} + m\Delta W_{22}^{(2)(t-1)},$$

$$W_{23}^{(2)(t)} = W_{23}^{(2)(t)} + \Delta W_{23}^{(2)(t)} + m\Delta W_{23}^{(2)(t-1)} = W_{23}^{(2)(t)} - \alpha\delta_1^{(3)}a_3^{(2)} + m\Delta W_{23}^{(2)(t-1)}$$

当误差传输到第 2 层时, 就不需要再往前传播了, 因为第 1 层是输入信号, 没有误差。

上面就是对于每个训练样本的输入信号前向传播、误差反向传输的全过程, 并且根据每层的误差调整了相应的连接权值。以上介绍部分, 是通过一个简单特定的网络来进行描述的, 主要是给读者一个感性认识, 知道具体怎么来进行相应的计算。在 4.2.3 节, 我们将针对更一般的情况, 推导出前馈网络输入信号正向传播、误差反向传播、根据误差调整相应连接权值的公式, 读者可以参考本节来加深理解。

4.2.3 算法推导

为了推导方便, 下面进行以下符号约定。

假设前馈神经网络由 L 层组成, 用上标 l 来代表层数, 则 $l \in \{1, 2, \dots, L\}$, 其中第 L 层为输出层, 每层的神经元数用 n_l 表示。

下标 i 表示第 i 个神经元, 下标 j 表示前一层神经元。

训练样本集中的训练样本表示形式为: $(\mathbf{x}^{(s)}, \mathbf{y}^{(s)})$, 其中样本总数为 S 。

神经元输入用 $z_i^{(l)}$ 表示, 其中 l 代表第 l 层, i 代表第 l 层第 i 个神经元。

神经元输出用 $a_i^{(l)}$ 表示, 其中 l 代表第 l 层, i 代表第 l 层第 i 个神经元。

层间连接权值用矩阵 $\mathbf{W}^{(l)}$ 表示, 代表从第 l 层指向第 $l+1$ 层。

当权值调整时, $W_{ij}^{(l)(t)}$ 表示从第 l 层的 j 节点, 指向第 $l+1$ 层的第 i 个节点, 在第 t 次调整时的值。

误差用 $\delta_i^{(l)}$ 表示, 代表第 l 层第 i 个节点的误差。

对于任意一个训练样本 $(\mathbf{x}^{(s)}, \mathbf{y}^{(s)})$, 第 1 层为输入层, 可以直接从训练样本中得到其输出:

$$a_i^{(1)} = x_i^{(s)} \quad i \in \{0, 1, 2, \dots, l_1\}$$

我们的讨论从第 2 层开始, 也就是第一个隐藏层, 对 $2 \leq l \leq L-1$, 即所有隐藏层, 其神经元数量为 n_l , 输入为:

$$z_i^{(l)} = \sum_{j=0}^{n_{l-1}} W_{ij}^{(l-1)} a_j^{(l-1)}$$

输出为:

$$a_i^{(l)} = f_h(z_i^{(l)})$$

上述过程一直重复到第 $L-1$ 层, 即最后一个隐藏层, 对于输出层而言, 有两种比较常见的形式, 对于两个类别的判断问题, 输出层只有一个以 Sigmoid 函数为激活函数的神经元,

其神经元输出可以视为该类别出现的概率。如果对于多类别（ $K \geq 3$ ），输出层则由 K 个 softmax 为激活函数的神经元中出现概率最大的神经元作为判断的类别。后文将分别讨论这两种情况。

1. 两类别判断

输出层神经元的输入值计算与隐藏层相同，如下：

$$z_1^{(L)} = \sum_{j=0}^{n_{L-1}} W_{1j}^{(L-1)} a_j^{(L-1)}$$

由于输出层只有一个神经元，其激活函数为 Sigmoid 函数，用 σ 表示，其输出为：

$$a_1^{(L)} = f_o(z_1^{(L)}) = \sigma(z_1^{(L)})$$

因为是二分类问题，因此输出层神经元的输出，可以理解为 $c=1$ 时的概率，因为只能有两种类别可以选择，因此输出层的概率分布为伯努利分布 $\text{Bernoulli}(\phi) = \phi$ ，用 c 来表示输出值所属类别： $c \in \{0,1\}$ ，则有如下公式：

$$p(c=1|\mathbf{a}^{(L-1)}; \mathbf{W}^{(L-1)}) = a_1^{(L)} = \sigma(z_1^{(L)}) \quad (1)$$

$$p(c=0|\mathbf{a}^{(L-1)}; \mathbf{W}^{(L-1)}) = 1 - a_1^{(L)} = 1 - \sigma(z_1^{(L)}) \quad (2)$$

式中， $\mathbf{a}^{(L-1)}$ 为最后隐藏层的输出向量， $\mathbf{a}^{(L-1)} \in \mathbb{R}^{n_{L-1}+1}$ ， $\mathbf{W}^{(L-1)}$ 为最后隐藏层到输出层的连接权值矩阵。

可以将上述两式进行合并：

$$p(c|\mathbf{a}^{(L-1)}; \mathbf{W}^{(L-1)}) = (a_1^{(L)})^c (1 - a_1^{(L)})^{1-c}$$

当 $c=1$ 时，等号右侧第二项指数为 0，其值为 1，就变为式（1）。当 $c=0$ 时，等号右边第一项指数为 0，其值为 1，就变为式（2）。

将负对数似然函数定义为代价函数（损失函数），则：

$$\mathcal{L}(\mathbf{a}^{(L-1)}, c) = -\log(p(c|\mathbf{a}^{(L-1)}; \mathbf{W}^{(L-1)})) = ((1-c)\log(1 - a_1^{(L)}) - c\log(a_1^{(L)}))$$

问题就转化为通过调整多层感知器模型参数，使负对数似然函数最小的问题。可以通过梯度下降算法来达到这一目的。

在这里要提一下，在当前很多教科书中，在讲多层感知器模型学习算法时，是将输出层的最小平方误差 $\text{MSE} = \|\mathbf{y}^{(s)} - \mathbf{a}_1^{(L)}\|^2$ 作为代价函数，通过调整连接权值，使其达到最小的。

但是直接对最小平方误差公式中的连接权值进行求导，应用梯度下降算法调整权值时，

由于输出层神经元的激活函数为 Sigmoid 函数，在输入值过大或过小时，会出现饱和现象，此时的梯度会非常小，造成学习效率很低，这也在一定程度上造成了多层感知器模型在 10 年前逐渐让位于支撑向量机。

当采用负对数似然函数作为代价函数，对连接权值进行求导时，应用梯度下降算法调整权值可以有效地避免这一问题，这可以视为多层感知器模型在过去一二十年间最重要的进展之一。另一进展是隐藏层采用 ReLU 神经元，而不是 Sigmoid 或双曲正切神经元。正是由于算法层面的这些进展，才使得古老的 BP 算法在当前深度学习时代，重新得到了研究人员的重视。

若想求负对数似然函数对于连接权值的偏导数，可以采用链式求导法则，先求负对数似然函数对输出层输出的导数：

$$\begin{aligned}\frac{\partial \mathcal{L}(\mathbf{a}^{(L-1)}, c)}{\partial a_1^{(L)}} &= \frac{\partial}{\partial a_1^{(L)}} \left((1-c) \log(1 - a_1^{(L)}) - c \log(a_1^{(L)}) \right) \\ &= \frac{1-c}{1-a_1^{(L)}} - \frac{c}{a_1^{(L)}} = \frac{(1-c)a_1^{(L)} - c(1-a_1^{(L)})}{a_1^{(L)}(1-a_1^{(L)})} = \frac{a_1^{(L)} - c}{a_1^{(L)}(1-a_1^{(L)})}\end{aligned}$$

接下来求输出层输出对最后隐藏层输出的导数：

$$\begin{aligned}\delta_1^{(L)} &= \frac{\partial \mathcal{L}(\mathbf{a}^{(L-1)}, c)}{\partial z_1^{(L)}} = \frac{\partial \mathcal{L}(\mathbf{a}^{(L-1)}, c)}{\partial a_1^{(L)}} \cdot \frac{\partial a_1^{(L)}}{\partial z_1^{(L)}} \\ &= \frac{a_1^{(L)} - c}{a_1^{(L)}(1-a_1^{(L)})} \cdot \frac{\partial \sigma(z_1^{(L)})}{\partial z_1^{(L)}} = \frac{a_1^{(L)} - c}{a_1^{(L)}(1-a_1^{(L)})} (a_1^{(L)}(1-a_1^{(L)})) \\ &= a_1^{(L)} - c\end{aligned}$$

可以注意到， $c \in \{0,1\}$ ，其实际就是训练样本 $(\mathbf{x}^{(s)}, \mathbf{y}^{(s)})$ 中的 $\mathbf{y}^{(s)}$ ，也是前面的简单网络示例中输出层误差直接用 $\mathbf{a}_1^{(L)} - \mathbf{y}^{(s)}$ 来求解的原因。

由此可见，上式中定义的就是输出层的误差。有了误差公式之后，就可以运行误差反向传播算法了。

下面先讨论在输出层有不止一个神经元的情况下，怎样求出输出层误差，然后统一讲解误差反向传播和权值调整问题。

2. 多类别判断

如果处理的问题类别 $K \geq 3$ ，输出层神经元数目就为 $n_L = K$ ，此时输出层的分布将为多项式分布，输出层的输出为：

$$p(c = i | \mathbf{a}^{(L-1)}; \mathbf{W}^{(L-1)}) = a_i^{(L)} = \frac{e^{z_i^{(L)}}}{\sum_{k=1}^K e^{z_k^{(L)}}}$$

可以视为每个类别出现的概率，网络的输出取所有输出层神经元输出值最大的一个，输出值为 1，其余神经元输出值为 0，例如当 $n_L = 5$ 时，第 2 个类别上式所对应的值最大，即其出现的概率最大，用向量 $\mathbf{c} = [0 \ 1 \ 0 \ 0 \ 0]^T$ 来表示，对应的似然函数则可以定义为：

$$p(\mathbf{c} | \mathbf{a}^{(L-1)}; \mathbf{W}^{(L-1)}) = \prod_{k=1}^{n_L} (a_k^{(L)})^{c_k}$$

负对数似然函数可以定义为：

$$\mathcal{L}(\mathbf{a}^{(L-1)}, \mathbf{c}) = -\log \left(\prod_{k=1}^{n_L} (a_k^{(L)})^{c_k} \right) = -\sum_{k=1}^{n_L} c_k \log(a_k^{(L)})$$

接下来的任务是通过调整连接权值，使负对数似然函数达到最小，从而最终使多层感知器模型输出误差值达到最小。所以需要对上式求偏导，由于直接对连接权值求导比较困难，故应用链式求导法则，首先对输出层输出求偏导，由于输出层由多个神经元组成，下面对第 i 个神经元求偏导：

$$\frac{\partial \mathcal{L}(\mathbf{a}^{(L)}, \mathbf{c})}{\partial a_i^{(L)}} = -\frac{c_i}{a_i^{(L)}}$$

下面求负对数似然函数对输出层输入量的偏导：

$$\frac{\partial \mathcal{L}(\mathbf{a}^{(L)}, \mathbf{c})}{\partial z_k^{(L)}} = \sum_{i=1}^{n_L} \frac{\partial \mathcal{L}(\mathbf{a}^{(L-1)}, \mathbf{c})}{\partial a_i^{(L)}} \cdot \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} = \sum_{i=1}^{n_L} \left(-\frac{c_i}{a_i^{(L)}} \right) \cdot \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}}$$

经过推导可得：

$$\delta_i^{(L)} = \frac{\partial \mathcal{L}(\mathbf{a}^{(L-1)}, \mathbf{c})}{\partial z_i^{(L)}} = a_i^{(L)} - c_i$$

即输出层每个神经元误差的计算公式。

3. 误差反向传播

在得到输出层每个神经元的误差计算公式之后，就可以将误差反向传播，一直传输到第 2 层，即第一个隐藏层，因为第 1 层是输入层，其输出值就是输入信号，没有误差，所以不用计算。

对于隐藏层误差，将从 $l=L-1$ 开始，一直计算到 $l=2$ 层为止，先以 $l=L-1$ 最后一个隐藏层为例，推导出误差计算公式：

$$\delta_i^{(l)} = \frac{\partial \mathcal{L}(\mathbf{a}^{(L)}, c)}{\partial z_i^{(l)}} = \frac{\partial \mathcal{L}(\mathbf{a}^{(L)}, c)}{\partial a_i^{(l)}} \cdot \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} \quad (1)$$

$$= \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} \cdot \sum_{j=1}^{n_L} \frac{\partial \mathcal{L}(\mathbf{a}^{(L)}, c)}{\partial z_j^{(L)}} \cdot \frac{\partial z_j^{(L)}}{\partial a_i^{(l)}} \quad (2)$$

$$= f_h' \left(z_i^{(l)} \right) \cdot \sum_{j=1}^{n_L} \delta_j^{(L)} \cdot W_{ji}^{(l)}$$

在①处，要想继续运用链式求导法则，必须跟踪 $z_i^{(l)}$ 输出信号传入输出层的所有神经元，因为第 l 层第 i 个神经元的误差会影响这些神经元。对于每个输出层神经元，先用负对数似然函数对该神经元输入信号求导，然后再用该神经元输入信号对第 l 层第 i 个神经元输出求导，如②所示。对其进行化简，上述两个公式正好是输出层对应神经元的误差，以及第 l 层第 i 个神经元到相应输出层神经元的连接权值。

此后，可以将 $L-1$ 层视为“输出层”，重复上面的推导过程，就可以求出所有隐藏层 ($l \geq 2$) 的误差了。

所以第 l 层通用误差计算公式为：

$$\delta_i^{(l)} = f_h' \left(z_i^{(l)} \right) \cdot \sum_{j=1}^{n_{l+1}} \delta_j^{(l+1)} \cdot W_{ji}^{(l)}$$

求出每层每个神经元的误差之后，就可以逐层调整权值，下面将讲述权值的调整公式。

4. 权值调整

求出每个神经元的误差之后，需要根据误差值对权值进行调整，首先求出负对数似然函数相对于该神经元连接权值的导数。

根据定义，第 l 层第 i 个神经元的误差为：

$$\begin{aligned} \frac{\partial \mathcal{L}(\mathbf{a}^{(L)}, c)}{W_{ij}^{(l-1)}} &= \frac{\partial \mathcal{L}(\mathbf{a}^{(L)}, c)}{\partial z_i^{(l)}} \cdot \frac{\partial z_i^{(l)}}{W_{ij}^{(l-1)}} \\ &= \delta_i^{(l)} a_j^{(l-1)} \end{aligned} \quad (1)$$

在①处，先求负对数似然函数对第 l 层第 i 个神经元输入信号的导数，然后再求输入信号对连接权值的导数，第一项正好是第 l 层第 i 个神经元定义的误差值，而第二项则是 $l-1$ 层神经元向第 l 层第 i 个神经元的输出值。

求出每个神经元对连接权值的导数之后，就可以根据梯度下降算法调整该神经元相关

的连接权值了。

定义在第 r 轮的连接权值调整中权值调整量为：

$$\Delta W_{ij}^{(l)(r)} = -\alpha \frac{\partial \mathcal{L}}{\partial W_{ij}^{(l)(r)}}$$

式中， α 为学习率，控制多层感知器学习算法收敛速度，通常取 $\alpha \in (0,1)$ ，例如 $\alpha = 0.1$ 。每调整一次权值，算作一轮， r 的值加 1。

直接使用梯度下降算法，容易陷入局部最小点，因为在局部最小点，导数的值为 0，权值将不再进行调整，因此也就无法走出局部最小值点了。

为了克服上述缺点，人们在权值调整时引入了动量项，其权值调整量为：

$$\Delta W_{ij}^{(l)(r)} = m \Delta W_{ij}^{(l)(r-1)} - \alpha \frac{\partial \mathcal{L}}{\partial W_{ij}^{(l)(r)}}$$

式中， m 为动量项且 $m \in [0,1]$ ，用其乘以上一次权值调整的值作为权值调整的惯性，来协助算法逃出局部极小值点。

4.3 向量化表示形式

在 4.2 节的多层感知器模型算法描述中，采用的是数字点积的方式，这种方式的优点是直观、易于理解，但缺点是实现效率较低。在实际应用的神经网络中，都用矩阵运算来进行加速，因为目前有很多成熟的矩阵科学运算库，经过专家多年优化，计算效率非常高，值得采用。在本节中，我们将介绍多层感知器模型学习算法的矩阵表示形式。

要研究的多层感知器模型如图 4.5 所示。

对于输入信号，其为 $\mathbf{x}^{(s)} \in \mathbb{R}^{n_1}$ ，为 n_1 维向量， s 代表第 s 个训练样本， n_1 是输入向量维度，同时也是输入层神经元数目。

对于每一个训练样本，首先使输入信号等于输入层的输出，即：

$$\mathbf{a}_{\text{raw}}^{(1)} = \mathbf{x}^{(s)} \quad \mathbf{a}_{\text{raw}}, \mathbf{x} \in \mathbb{R}^{n_1}$$

在向量 \mathbf{a}_{raw} 最前面加上 $a_0^{(1)} = 1$ 元素，形成最终输出向量 \mathbf{a} ：

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_{n_1} \end{bmatrix}, \quad \mathbf{a}_{\text{raw}} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_{n_1} \end{bmatrix}, \quad \mathbf{a} = \begin{bmatrix} a_0^{(1)} = 1 \\ a_1^{(1)} = x_1 \\ a_2^{(1)} = x_2 \\ \dots \\ a_{n_1}^{(1)} = x_{n_1} \end{bmatrix}$$

从第2层开始,都需要先从前一层的输出经过线性组合形成输入量 z ,然后经过激活函数产生输出向量 \mathbf{a}_{raw} ,加上 $\mathbf{a}_0 = 1$ 分量后传输给下一层。所以下面的讨论以普通的第 l 层为例,其中 $l \in \{2, 3, \dots, L\}$ 。

第 $l-1$ 层到第 l 层的连接权值矩阵为

$$\mathbf{W}^{(l-1)} = \begin{bmatrix} W_{10}^{(l-1)} & W_{11}^{(l-1)} & \dots & W_{1n_{l-1}}^{(l-1)} \\ W_{20}^{(l-1)} & W_{21}^{(l-1)} & \dots & W_{2n_{l-1}}^{(l-1)} \\ \dots & \dots & \dots & \dots \\ W_{n_l 0}^{(l-1)} & W_{n_l 1}^{(l-1)} & \dots & W_{n_l n_{l-1}}^{(l-1)} \end{bmatrix}, \quad \mathbf{W}^{(l-1)} \in \mathbb{R}^{n_l \times (n_{l-1}+1)}$$

式中, $W_{ij}^{(l-1)}$ 代表第 $l-1$ 层第 j 个节点指向第 l 层第 i 个神经元节点。

首先来看第 l 层的输入向量计算:

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l-1)} \mathbf{a}^{(l-1)} = \begin{bmatrix} W_{10}^{(l-1)} & W_{11}^{(l-1)} & \dots & W_{1n_{l-1}}^{(l-1)} \\ W_{20}^{(l-1)} & W_{21}^{(l-1)} & \dots & W_{2n_{l-1}}^{(l-1)} \\ \dots & \dots & \dots & \dots \\ W_{n_l 0}^{(l-1)} & W_{n_l 1}^{(l-1)} & \dots & W_{n_l n_{l-1}}^{(l-1)} \end{bmatrix} \cdot \begin{bmatrix} a_0^{(l-1)} \\ a_1^{(l-1)} \\ a_2^{(l-1)} \\ \dots \\ a_{n_{l-1}}^{(l-1)} \end{bmatrix} = \begin{bmatrix} z_1^{(l)} \\ z_2^{(l)} \\ \dots \\ z_{n_l}^{(l)} \end{bmatrix}, \quad \mathbf{z}^{(l)} \in \mathbb{R}^{n_l}$$

求出各神经元的输入之后,可以求出各神经元的输出,以第 l 层为例:

$$\mathbf{ar}^{(l)} = \mathbf{f}_h(\mathbf{z}^{(l)}) = \begin{bmatrix} f_h(z_1^{(l)}) \\ f_h(z_2^{(l)}) \\ \dots \\ f_h(z_{n_l}^{(l)}) \end{bmatrix} = \begin{bmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \dots \\ a_{n_l}^{(l)} \end{bmatrix}$$

再加上第0个神经元,组成第 l 层的输出向量:

$$\mathbf{a}^{(l)} = \begin{bmatrix} a_0^{(l)} \\ a_1^{(l)} \\ \dots \\ a_{n_l}^{(l)} \end{bmatrix}$$

完成以上步骤后,就可以进行第 $l+1$ 层的计算了,方法与上面的方法一样,一直进行到第 L 层输出层为止,就完成了整个多层感知器模型信号前向传播的过程。

4.4 应用要点

我们已经向读者介绍了多层感知器模型的学习算法,以及详细的推导过程,并且介绍了用向量和矩阵表示前向传播算法,以提高计算效率。

掌握了这些知识之后，是否就能设计出好的多层感知器模型了呢？答案可能让大家失望了。除掌握这些基本知识之外，还需要针对所研究问题找出可以描述该问题的最佳特征，准备更具代表性的训练样本集，设计合适的网络架构，例如由几层网络组成、每层神经元的数量、选择哪种激活函数等。对于这些问题，目前还没有科学的解决方案，只有一些经验可供参考，因此可以说多层感知器模型的应用更像一门艺术，而不是工程技术应用。

尽管如此，在应用多层感知器模型时，还是有一些指南供我们参考的，本节就列出这些非常易于遵守且具有显著效果的实践指南。

我们知道，输入层、隐藏层和输出层组成的三层感知器模型，可以以任意精度拟合任意函数，这就是著名的万能逼近理论。但是在实际应用中，会通过连接权值调整来达到这个目的。可以设想连接权值是一个空间中非常不平坦的曲面，具有无数个坑坑洼洼的局部最小值点，即一个个小坑，梯度下降算法就好像我们拿一个小球放到这个曲面上，小球有非常大的概率落到某个局部最小值点，而不是全局最小值点。对于这个问题，虽然可以通过在线学习的随机梯度下降算法和权值调整中引入动量项来部分解决，但是不能完全避免。

所以在这里只讨论几个可以显著提高多层感知器泛化能力的方法。所谓泛化能力，就是神经网络遇到从来没见过的样本，同样可以做出正确判断的能力，在实际应用中就是解决欠拟合和过拟合问题。欠拟合就是由于训练样本过少、训练时间过短或模型太简单，例如采用线性分类器解决复杂的非线性分类问题，模型在测试样本集上的误差过大的情况。过拟合就是模型在训练样本集上有非常完美的表现，但是在测试样本集上，因为之前没有见过，却表现不佳的情况。

对于欠拟合情况，可以通过采集更多的训练样本集，或者更换更加复杂的模型来解决。但是过拟合情况就很难解决了，目前主要通过合适的调节（**Regulation**）来实现，这是当前深度学习领域的研究热点。下面将向读者介绍早期停止、输入信号调整、权值初始化等内容。

4.4.1 输入信号模型

选择合适的特征来描述所研究的问题，是应用深度学习算法成功的关键。事实上，如果可以找到合适的特征来描述问题，深度学习的应用范围和效果将极大地提高。但是，一方面，我们不知道用什么样的特征来描述问题，例如在数字图像识别中，到底选择什么作为特征；另一方面，即使我们知道采用什么特征，但是特征的提取可能要花费大量的人力、物力和时间，我们也负担不起。在这里只想强调一点，掌握所研究问题的专业知识比掌握深度学习算法更重要，所以特征选择的优先级要优于深度学习算法的选择。

虽然特征选择因所研究问题而异，没有公认有效的方法论作为指导，但是还是可以对输入信号进行相应的处理，使其更容易收敛，提高算法的性能。在这里要向读者介绍的是输入信号的标准化，即把输入信号先转化为数学期望为 0 且方差为 1 的训练样本集。1998 年，LeCun 证明，经过上述变换后，输入信号的性质不变，但是通过把输入信号压缩到常

用的神经元激活函数的有效区域，可以极大地提高算法的性能。

首先计算输入向量第 i 个分量的期望：

$$m_i = \frac{1}{|S|} \sum_{s \in S} x_i$$

然后计算输入向量第 i 个分量的方差：

$$\sigma_i = \sqrt{\frac{1}{|S|} \sum_{s \in S} (x_i - m_i)^2}$$

对于输入向量第 i 个分量，其调整后的值为：

$$\tilde{x}_i = \frac{x_i - m_i}{\sigma_i}$$

将新的输入向量用于训练过程，可以显著提高算法的性能，而且不会丢失输入信号中的信息，是值得尝试的方法。

4.4.2 权值初始化

需要在算法开始前，用比较小的随机数初始化所有连接权值，例如取 $[-0.1, 0.1]$ 等都是比较合理的选择。在使用 Theano 框架时，无论使用均匀分布，还是使用期望为 0、方差为 0.1 的高斯分布，都可以取得较好的效果。

4.4.3 早期停止

早期停止是一种提高模型泛化能力的方法。具体方法是将训练样本集分为两部分，一部分作为训练样本集，然后拿出一小部分（如 10% 左右）作为验证样本集。我们用训练样本集训练模型并调整连接权值。每隔一定时间，运行验证样本集，求出在验证样本集上的误差值。如果这个误差值持续减小，就一直运行训练过程。当验证样本集上的误差值不再减小，甚至开始增大时，就停止训练过程。这时运行测试样本集，求出测试样本集上的误差值，作为模型的性能标准。

早期停止是一种简单却行之有效的提高模型泛化能力的方法，但是在训练样本集本来就很小的情况下，由于要拿出一部分样本来作为验证样本集，训练样本集就会变得更小，这将严重影响模型的泛化能力。解决这一问题的方法是，首先随机拿出一小部分（如 10%）训练样本集作为验证样本集，对网络进行训练，然后再随机拿出一小部分（如 10%）作为验证样本集对模型进行训练，这样循环进行多次，以达到较好的应用效果。

4.4.4 输入信号调整

神经网络泛化能力不强的一个根本原因是训练样本集过小或训练样本集的代表性不够强，如果可以增大训练样本集的数量，将可以大大提高模型的泛化能力。但是由于实际情况的限制，可能无法取到更大的训练样本集。为了解决这一问题，研究人员发现，通过系统模拟手段，人为生成一些仿真的训练样本集，只要仿真程序足够好，同样可以极大地提高模型的泛化能力。

以 MNIST 手写数字识别为例，有两种方式来增加训练样本数量，如图 4.6 所示。

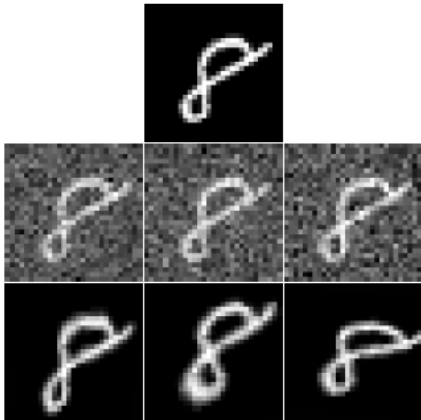


图 4.6 输入信号调整

在图 4.6 中，第 1 行为原始图片，第 2 行是在原始图片中加入了高斯噪声，第 3 行为对原始图片进行矢量化，并对图片进行拉伸等变换。读者可以思考一下，哪种方式对提高模型泛化能力更有效呢？

事实证明，第一种方式虽然增加了训练样本集数量，但是由于加入的是白噪声，神经网络可以很好地去掉这些白噪声，所以在神经网络看来，第 2 行的训练样本与第 1 行的原始图片是完全一样的，所以起不到任何效果。

第二种方式将图片进行了拉伸处理，完全模仿了手写数字的变化情况，有效地增加了训练样本数量，因此取得了非常好的效果。以多层感知器模型为例，同样是三层结构 784-800-10，在没有采用输入信号调整的情况下，误差率为 1.6%，而采用了输入信号调整之后，误差率降到 0.7%，由此可见这种方式的效果。

4.5 TensorFlow 实现 MLP

多层感知器模型由输入层、隐藏层和输出层组成，输入层只是简单地将训练样本赋给

输入层神经元作为输出，隐藏层处理输入信号，经过非线性激活函数产生输出，传输给下一个隐藏层，直到最后一个隐藏层将信号传递给输出层。在 MNIST 手写数字识别问题中，可以将输出层设计为采用多类别模式分类的逻辑回归模型。

在讲述具体的实现之前，我们先来定义一下变量的命名规则。我们用 a 来表示神经元输出，例如第 i 层第 j 个神经元的输出用 $a_{i,j}$ 来表示。第 i 层第 j 个神经元的输入用 $z_{i,j}$ 来表示。第 i 层第 j 个神经元的偏置值用 $b_{i,j}$ 来表示。从第 1 层第 i 个神经元到第 $la1$ 层第 j 个神经元的连接权值矩阵用 $W_{1,j,i}$ 。如果我们用 L_i 来表示第 i 层的神经元数量，则可以通过第 1 层神经元输出向量 a_1 来计算第 $la1$ 层神经元的输入向量 z_{la1} ，如下所示：

$$W^{(l)} a^{(l)} = \begin{bmatrix} W_{1,1}^{(l)} & W_{1,2}^{(l)} & \dots & W_{1,L_l}^{(l)} \\ W_{2,1}^{(l)} & W_{2,2}^{(l)} & \dots & W_{2,L_l}^{(l)} \\ \vdots & \vdots & \dots & \vdots \\ W_{L_{l+1},1}^{(l)} & W_{L_{l+1},2}^{(l)} & \dots & W_{L_{l+1},L_l}^{(l)} \end{bmatrix} \begin{bmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \vdots \\ a_{L_l}^{(l)} \end{bmatrix} = \begin{bmatrix} z_1^{(l+1)} \\ z_2^{(l+1)} \\ \vdots \\ z_{L_{l+1}}^{(l+1)} \end{bmatrix}$$

所以在下面的程序中，输入层作为第 1 层，隐藏层作为第 2 层，输出层作为第 3 层。第 1 层到第 2 层的连接权值矩阵用 W_1 来表示，而第 2 层的偏置值用 b_2 来表示，第 2 层的输入向量用 z_2 表示，输出值用 a_2 来表示，如果采用 Dropout 调整技术，则输出值为 $a_{2_dropout}$ ，依此类推。

有了上面的命名规范，我们就可以来看具体程序实现了。与网上大多数例子不同，在本章的例子中，除了向大家演示 TensorFlow 实现基本的多层感知器模型，还会向大家演示一些实际应用中的常用技术，包括：采用 Adam 优化（虽然目前没有公认最好的优化算法，但是 Adam 无疑是 2017 年学术界最热的优化算法）、采用 L2 调整项（权值衰减）、采用 Early Stopping 调整技术、采用 Dropout 调整技术。同时向大家展示了在隐藏层采用 Sigmoid 激活函数或 ReLU 激活函数，并且比较了二者的不同。

首先载入 MNIST 手写数字识别数据集，代码如下：

```
1 def load_datasets(self):
2     ''' 调用 TensorFlow 的 input_data，读入 MNIST 手写数字识别数据集的
3     训练样本集、验证样本集、测试样本集
4     '''
5     mnist = input_data.read_data_sets(self.datasets_dir,
6     one_hot=True)
7     X_train = mnist.train.images
8     y_train = mnist.train.labels
9     X_validation = mnist.validation.images
10    y_validation = mnist.validation.labels
11    X_test = mnist.test.images
12    y_test = mnist.test.labels
13    print('X_train: {0} y_train:{1}'.format(
14        X_train.shape, y_train.shape))
15    print('X_validation: {0} y_validation:{1}'.format(
16        X_validation.shape, y_validation.shape))
17    print('X_test: {0} y_test:{1}'.format(
18        X_test.shape, y_test.shape))
19    image_raw = (X_train[1] * 255).astype(int)
20    image = image_raw.reshape(28, 28)
```

```

21     label = y_train[1]
22     idx = 0
23     for item in label:
24         if 1 == item:
25             break
26         idx += 1
27     plt.title('digit:{0}'.format(idx))
28     plt.imshow(image, cmap='gray')
29     plt.show()
30     return X_train, y_train, X_validation, y_validation, \
31           X_test, y_test, mnist

```

第 4、5 行：调用 TensorFlow 的 `input_data` 的 `read_data_sets` 方法，第一个参数为数据集存放路径，第二个参数是标签集的格式。在原始 MNIST 数据集中，我们知道每个样本是 28×28 的黑白图片，对应的是 0~9 的数字标签，所以其格式为：[...784 (28×28) 像素点的值...][3]。其中，第一项为 784 (28×28) 个 0~1 的浮点数，0 代表黑色，1 代表白色；第二项的“3”代表这个样本是数字 3。为了后续处理方便，我们将标签[3]改为 one-hot 形式，因为标签代表 0~9 的数字，所以标签集为 10 维向量，每维上取值为 0 代表不是这个对应位置的数字，取值为 1 代表是这个对应位置的数字。其中，只有一维可以取 1，因此称之为 one-hot，还以上面的例子为例，标签集的格式就变为：[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]，因为第四位为 1，所以代表这个样本是数字 3。

第 7 行：取出训练样本集输入信号集 `X_train`，其为设计矩阵形式，每一行代表一个样本，行数为训练样本集中的样本数量。在这个例子中，就 $X_{\text{train}} \in \mathbb{R}^{55000 \times 784}$ ，其中训练样本集中有 55000 个样本，每个样本是 784 (28×28) 维的图片。

第 8 行：取出训练样本集标签集 `y_train`，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为训练样本集中的样本数量。在这个例子中，就 $y_{\text{train}} \in \mathbb{R}^{55000 \times 10}$ ，其中训练样本集中有 55000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 9 行：取出验证样本集输入信号集 `X_validation`，其为设计矩阵形式，每一行代表一个样本，行数为验证样本集中的样本数量。在这个例子中，就 $X_{\text{validation}} \in \mathbb{R}^{5000 \times 784}$ ，其中验证样本集中有 5000 个样本，每个样本是 784 (28×28) 维的图片。根据前面我们的讨论可以知道，在训练过程中，为了防止模型出现过拟合，模型的泛化能力降低（模型在训练样本集达到非常高的精度，但是在未见过的测试样本集或实际应用中，精度反而不高），通常会采用 **Early Stopping** 策略，就是在逻辑回归模型训练过程中，只用训练样本集对模型进行训练，每隔一定的时间间隔，计算模型在未见过的验证样本集上识别的精度，并记录迄今为止在验证样本集上取得的最高精度。我们会发现，在训练初期，验证样本集上的识别精度会稳步提高，但是到了一定阶段之后，验证样本集上的识别精度就不会再明显提高了，甚至开始逐渐下降，这就说明模型出现了过拟合，这时就可以停止模型训练，将在验证样本集上取得最佳识别精度的模型参数作为模型最终的参数。综上所述，验证样本集主要用于防止模型出现过拟合，为 **Early Stopping** 算法提供终止依据。

第 10 行：取出验证样本集标签集 `y_validation`，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为训练样本集中的样本数量。在这个例子中，就 $y_{\text{validation}} \in \mathbb{R}^{5000 \times 10}$ ，其中验证样本集中有 5000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

$R^{5000 \times 10}$ ，其中验证样本集中有 5000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 11 行：取出测试样本集输入信号集 X_{test} ，其为设计矩阵形式，每一行代表一个样本，行数为训练样本集中的样本数量。在这个例子中，就 $X_{\text{test}} \in R^{10000 \times 784}$ ，其中训练样本集中有 10000 个样本，每个样本是 784 (28×28) 维的图片。测试样本集主要用于模型训练结束后对模型性能进行评估。由于模型没有见过测试样本集中的样本，可以模拟模型在实际部署之后的情况，模型在测试样本集上的识别精度，基本可以视为模型在实际应用中可以达到的精度。

第 12 行：取出测试样本集标签集 y_{test} ，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为测试样本集中的样本数量。在这个例子中，就 $y_{\text{test}} \in R^{10000 \times 10}$ ，其中测试样本集中有 10000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 13~18 行：分别打印训练样本集、验证样本集、测试样本集的内容。

第 19 行：以训练样本集中第 2 个样本为例，带领大家看一下 MNIST 手写数字识别数据集的具体内容。我们首先读出样本输入信号，其为 28×28=784 维向量，元素为 0~1 之间的浮点数，由于要进行显示，所以将每个元素乘以 255，变为 0~255 的浮点数，然后再将其变为 0~255 的整数。

第 20 行：将 784 维向量 `image_raw` 转换为 28×28 的矩阵。

第 21 行：取出该样本的正确结果标签。

第 21 行：定义索引号。

第 23 行：对 one-hot 形式标签循环第 24~26 行操作。

第 24、25 行：如果标签 one-hot 向量此元素为 1，则终止循环，此时 `idx` 的值就是所对应的数字。

第 26 行：如果标签 one-hot 向量此元素不为 1，则索引号加 1。

第 27 行：将样本标签的正确结果显示在图片标题中。

第 28 行：以灰度图像形式显示样本对应的图片。

第 29 行：具体显示图片。

第 30、31 行：返回训练样本集输入信号集、训练样本集标签集、验证样本集输入信号集、验证样本集标签集、测试样本集输入信号集、测试样本集标签集。

运行上面的程序，后台打印结果如图 4.7 所示。

```
X_train: (55000, 784) y_train:(55000, 10)
X_validation: (5000, 784) y_validation:(5000, 10)
X_test: (10000, 784) y_test:(10000, 10)
```

图 4.7 载入 MNIST 数据集的结果

对应的图像如图 4.8 所示。

从图 4.8 中可以看出，对于我们来说，这张图也是比较难以认出的，但是在后面我们将看到，我们的多层感知器模型在没有见到的测试样本集上的正确率可以达到 98% 以上，精度还是非常高的，由此可见，即使只有一层的多层感知器模型，其识别能力也是非常强大的。

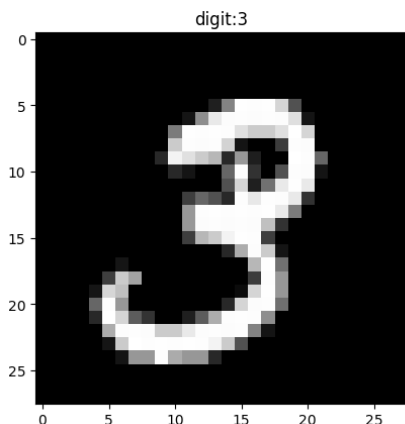


图 4.8 MNIST 数据集样本图像

需要注意的是，上面的代码是为了让大家对 MNIST 手写数字识别数据集有一个感性的认识，所以加上了第 13~29 行的代码，在实际运行过程中，读者应该将这些代码注释掉或者直接删除，这样才能高效地运行 `train` 方法和 `run` 方法。

下面我们来重点讲述多层感知器模型的构建方法。多层感知器模型在设计方面主要需要考虑以下一些因素。

- ❑ 模型的深度：隐藏层的数量。
- ❑ 模型的宽度：每个隐藏层包含神经元的数量。
- ❑ 隐藏层神经元类别：可以选择 Sigmoid、tanh、ReLU 这三种常见类型。
- ❑ 代价函数类型：交叉熵（Cross Entropy）、最小平方误差（MSE）。
- ❑ 优化算法选择：随机梯度下降、Ada、RMSProp、Adam 等。
- ❑ 调整策略：L2 调整项（权值衰减）、Early Stopping、Dropout。
- ❑ 输出层：Sigmoid（二分类）或 Softmax（多分类）。

在这个例子中，为了简化问题，我们选择只有一个隐藏层，隐藏层神经元数量为 512。隐藏层神经元类型我们给出了两种选择，一种是 Sigmoid 类型，另一种是 ReLU 类型，通过比较这两种神经元类型，我们可以证明，ReLU 神经元在收敛速度和在测试样本集上的识别精度方面，都要优于 Sigmoid 神经元。所以在多层感知器模型中，一般情况下应该默认选择 ReLU 神经元。在代价函数方面，我们选择交叉熵函数，因为交叉熵函数所基于的负对数似然函数，可以有效地解决最小平方误差遇到的激活函数处于饱和区时梯度消失的问题，使基于梯度下降算法的效率更高。在优化算法方面，我们向大家演示了使用随机梯度下降和 Adam 两种算法。在调整策略方面，我们使用 L2 调整项（权值衰减）、Early Stopping、Dropout 这三种常见的调整策略。由于是多分类问题，输出层采用 Softmax 神经元。

下面我们来看基于 Sigmoid 神经元的模型构建，代码如下：

```
1 def build_sigmoid(self):
2     print('##### sigmoid #####')
3     self.keep_prob = 0.90
4     X = tf.placeholder(tf.float32, [None, 784])
5     y = tf.placeholder(tf.float32, shape=[None, 10])
6     keep_prob = tf.placeholder(tf.float32) #Dropout 失活率
```

```

7      # 隐藏层
8      W_1 = tf.Variable(tf.random_normal(shape=[784, 512], mean=0.0,
9      stddev=1.0)) # W_t
10     b_2 = tf.Variable(tf.zeros(shape=[512]))
11     z_2 = tf.matmul(X, W_1) + b_2
12     a_2 = tf.nn.sigmoid(z_2)
13     # 输出层
14     W_2 = tf.Variable(tf.random_normal(shape=[512, 10],
15     mean=0.0, stddev=1.0)) # W_t
16     b_3 = tf.Variable(tf.random_normal(shape=[10], mean=0.0,
17     stddev=1.0))
18     z_3 = tf.matmul(a_2, W_2) + b_3
19     y_ = tf.nn.softmax(z_3)
20     # 代价函数
21     # cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(y_, y)
22     cross_entropy = tf.reduce_sum(- y * tf.log(y_), 1)
23     loss = tf.reduce_mean(cross_entropy)
24     #train_step = tf.train.GradientDescentOptimizer(0.05).minimize(loss)
25     train_step = tf.train.AdamOptimizer(learning_rate=0.001, beta1=0.9,
26     beta2=0.999, epsilon=1e-08, use_locking=False,
27     name='Adam').minimize(loss)
28     # 精度计算
29     correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
30     accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
31     return X, y_, y, keep_prob, cross_entropy, train_step, \
32     correct_prediction, accuracy

```

第 3 行：由于要使用 Dropout 调整项，所以需要设置 Dropout 操作的参数。Dropout 是指在训练过程中，我们在每次样本学习时，随机地将 self.keep_prob 比例的隐藏层神经元的输出设置为 0，得到一个子网络，对这个子网络的参数进行学习。由于每次学习时选取的子网络是随机的，所以我们可以认为整体网络是由很多随机的子网络构成。在模型评估阶段，输入训练样本集样本时，我们不对隐藏层神经元进行选择，这时可以视为是很多随机的子网络同时运行，最后的识别结果是这些子网络投票的结果。由于采用 Sigmoid 神经元，需要去掉的隐藏层神经元比例较低，因此在这里进行单独设置。

第 4 行：定义存放输入信号集的 placeholder 类型的 X，每一行代表一个样本，维度为 $28 \times 28 = 784$ 维，一次会处理一个迷你批次，因此 X 的第一维为 None，在程序实际运行时，会根据迷你批次的大小给第一维赋上实际的值。

第 5 行：定义存放正确标签结果集的 placeholder 类型的 y，每一行代表一个样本的正确结果，维度为 10，代表 0~9 的数字，我们一次会处理一个迷你批次，因此 y 的第一维为 None，在程序实际运行时，会根据迷你批次的大小给第一维赋上实际的值。

第 6 行：定义 Dropout 时隐藏层神经元的保留率的 placeholder 类型变量，其值为一个百分比，代表我们会随机地将多少比例的隐藏层神经元的输出置为 0。

第 8、9 行：定义输入层到隐藏层的连接权值矩阵。注意：这里为了后续计算方便，实际上定义了我们在理论部分描述的连接权值矩阵的转置 $(W^{(1)})^T$ ，所以其维度为 784×512 。

我们使用均值为 0、标准差为 1.0 的正态分布随机数来初始化权值。

第 10 行：定义第 2 层，即隐藏层的偏置值 b_2，我们用 0 作为初始值。

第 11 行：求出第 2 层，即隐藏层的输入值。

第 12 行：求出第 2 层经过 Sigmoid 函数后的输出值。

第 14、15 行：定义第 2 层（隐藏层）到第 3 层（输出层）的连接权值矩阵，用均值为 0、标准差为 1.0 的正态分布随机数初始化连接权值。同样，为了计算方便，实际上定义的也是理论部分连接权值矩阵的转置($W^{(2)}$)^T。

第 16、17 行：初始化第 3 层（输出层）的偏置值，这里采用均值为 0、标准差为 1.0 的正态分布随机数进行初始化。

第 18 行：求出第 3 层（输出层）的输入值。

第 19 行：求出经过 Softmax 激活函数后的输出值，此值即每个类别的概率。

第 22 行：定义交叉熵。

第 23 行：将损失函数定义为交叉熵函数。

第 24 行：采用随机梯度下降算法，学习率为 0.05。

第 25~27 行：采用 Adam 优化算法。

第 29 行：利用 TensorFlow 的 argmax 函数，分别求出计算类别向量每个样本的最大值下标和类别向量每个样本的最大值下标，并对其进行比较。

第 30 行：求出预测精度，首先调用 TensorFlow 的 cast 函数，将第 16 行的结果变为浮点数列表，形式为：[1.0, 1.0, 0.0, 1.0, 1.0]，这里假设只有 5 个样本。再调用 TensorFlow 的 reduce_mean 函数求出这个列表的平均值：(1.0+1.0+0.0+1.0+1.0)/5=0.8，这个值就是模型预测的精度。

第 31、32 行：返回模型定义的 X, y_, y, keep_prob, cross_entropy, train_step, correct_prediction, accuracy。

我们将在后面介绍多层感知器模型的训练和运行方法，在这里先显示一下隐藏层采用 Sigmoid 神经元的训练情况，结果如图 4.9 所示。

在训练方法中，我们设置的训练样本集学习遍数为 10，但是上面的运行结果显示，在学习到第 7 遍时，由于 Early Stopping 调整项的作用，训练过程就终止了，此时我们在训练样本集上的识别精度为 97.6%，验证样本集上的识别精度为 93.1%，在测试样本集上的精度为 92.8%。读者可以尝试调整我们的调整项，可以使训练样本集上的识别精度达到 99% 左右，但是验证样本集的识别精度的提升就没有那么大了。

```
5:450# train:0.97079998254776, validation:0.9273999929428101
5:500# train:0.9674909114837646, validation:0.9259999990463257
6:0# train:0.9696132012557983, validation:0.9297999739646912
6:50# train:0.9719818234443665, validation:0.9296000003814697
6:100# train:0.9740545749664307, validation:0.9308000206947327
6:150# train:0.9745091199874878, validation:0.9300000071525574
6:200# train:0.97563636302948, validation:0.9294000267982483
6:250# train:0.9736363887786865, validation:0.928600013256073
6:300# train:0.9762545228004456, validation:0.9301999807357788
6:350# train:0.9760727286338806, validation:0.9315999746322632
6:400# train:0.9771090745925903, validation:0.9314000010490417
6:450# train:0.9756545424461365, validation:0.9318000078201294
6:500# train:0.9765272736549377, validation:0.9337999820709229
7:0# train:0.9789817929267883, validation:0.9308000206947327
7:50# train:0.9797272682189941, validation:0.9315999746322632
7:100# train:0.9816363453865051, validation:0.9340000152587891
7:150# train:0.9822545647621155, validation:0.930400013923645
7:200# train:0.9757636189460754, validation:0.9314000010490417
0.9279
```

图 4.9 Sigmoid 神经元训练结果

在训练样本集和验证样本集上的识别精度曲线如图 4.10 所示。

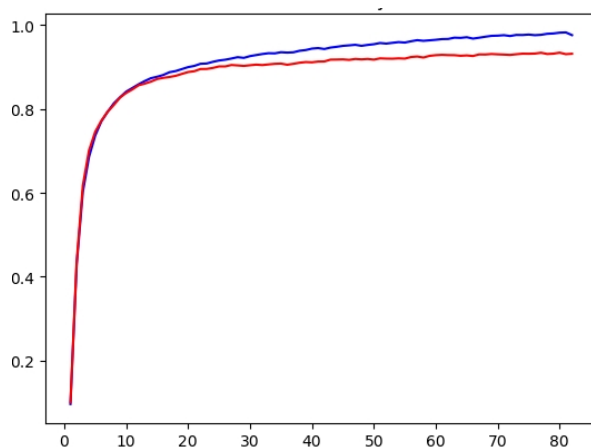


图 4.10 识别精度变化曲线

在上图中，深色曲线是训练样本集上的识别精度，浅色曲线是验证样本集上的识别精度。我们可以看到，在训练初期，训练样本集上的识别精度和验证样本集上的识别精度是基本重合的；而在训练后期，训练样本集上的识别精度会明显高于验证样本集上的识别精度，如果任由这种趋势发展下去，就可能出现过拟合现象，即虽然在训练样本集上的识别精度很高，但是在学习过程中没有见过的验证样本集和测试样本集上的识别精度却比较低。由于 Early Stopping 调整项的存在，我们及时停止训练过程，从而避免模型陷入过拟合（Overfitting）状态。

因为在隐藏层中采用 Sigmoid 神经元，虽然它在过去非常流行，但是现在已经基本不在实际应用中使用了，所以我们对 Sigmoid 神经元的讨论就到此为止了。下面将针对在隐藏层中采用 ReLU 神经元进行详细描述。

我们先来看在隐藏层采用 ReLU 神经元时模型的创建，代码如下：

```
1 def build_relu(self):
2     print('##### relu #####')
3     X = tf.placeholder(tf.float32, [None, 784])
4     y = tf.placeholder(tf.float32, [None, 10])
5     #隐藏层
6     W_1 = tf.Variable(tf.truncated_normal([784, 512], mean=0.0,
7         stddev=0.1)) #初始化隐含层权重 w1, 服从默认均值为 0,
8         #标准差为 0.1 的截断正态分布
9     b_2 = tf.Variable(tf.zeros([512])) #隐含层偏置 b1 全部初始化为 0
10    z_2 = tf.matmul(X, W_1) + b_2
11    a_2 = tf.nn.relu(z_2)
12    keep_prob = tf.placeholder(tf.float32) #Dropout 失活率
13    a_2_dropout = tf.nn.dropout(a_2, keep_prob)
14    #输出层
15    W_2 = tf.Variable(tf.zeros([512, 10]))
16    b_3 = tf.Variable(tf.zeros([10]))
17    z_3 = tf.matmul(a_2_dropout, W_2) + b_3
18    y_ = tf.nn.softmax(z_3)
19    #训练部分
```



```

20 cross_entropy = tf.reduce_mean(-tf.reduce_sum(y * tf.log(y_),
21     reduction_indices=[1]))
22 #train_step = tf.train.AdagradOptimizer(0.3).minimize(cross_entropy)
23 loss = cross_entropy + self.lanmeda*(tf.reduce_sum(W_1**2) +
24     tf.reduce_sum(W_2**2))
25 train_step = tf.train.AdamOptimizer(learning_rate=0.001, beta1=0.9,
26     beta2=0.999, epsilon=1e-08, use_locking=False,
27     name='Adam').minimize(loss)
28 correct_prediction = tf.equal(tf.argmax(y_, 1), tf.argmax(y, 1))
29 accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
30 self.saveModelTensor(X, y, W_1, b_2, z_2, a_2, W_2, b_3, z_3, y_,
31     cross_entropy, loss, train_step, correct_prediction, accuracy)
32 return X, y_, y, keep_prob, cross_entropy, train_step, \
33     correct_prediction, accuracy
34
35 def saveModelTensor(self, X, y, W_1, b_2, z_2, a_2, W_2, b_3, z_3, y_,
36     cross_entropy, loss, train_step, correct_prediction, accuracy):
37     # 保存模型
38     self.model['X'] = X
39     self.model['y'] = y
40     self.model['W_1'] = W_1
41     self.model['b_2'] = b_2
42     self.model['z_2'] = z_2
43     self.model['a_2'] = a_2
44     self.model['W_2'] = W_2
45     self.model['b_3'] = b_3
46     self.model['z_3'] = z_3
47     self.model['y_'] = y_
48     self.model['cross_entropy'] = cross_entropy
49     self.model['loss'] = loss
50     self.model['train_step'] = train_step
51     self.model['correct_prediction'] = correct_prediction
52     self.model['accuracy'] = accuracy

```

第 3 行：定义存放输入信号集的 `placeholder` 类型的 `X`，每一行代表一个样本，维度为 $28 \times 28 = 784$ 维，一次会处理一个迷你批次，因此 `X` 的第一维为 `None`，在程序实际运行时，会根据迷你批次的大小给第一维赋上实际的值。

第 4 行：定义存放正确标签结果集的 `placeholder` 类型的 `y`，每一行代表一个样本的正确结果，维度为 10，代表 0~9 的数字，我们一次会处理一个迷你批次，因此 `y` 的第一维为 `None`，在程序实际运行时，会根据迷你批次的大小给第一维赋上实际的值。

第 6~8 行：定义输入层到隐藏层的连接权值矩阵。注意：这里为了后续计算方便，实际上定义了我们在理论部分描述的连接权值矩阵的转置 $(W^{(1)})^T$ ，所以其维度为 784×512 。我们使用均值为 0、标准差为 1.0 的正态分布随机数来初始化权值。

第 9 行：定义第 2 层（隐藏层）的偏置值 `b_2`，我们用 0 作为初始值。

第 10 行：求出第 2 层（隐藏层）的输入值。

第 11 行：求出第 2 层经过 ReLU 函数后的输出值。

第 12 行：定义 Dropout 时隐藏层神经元的保留率的 `placeholder` 类型变量，其值为一个百分比，代表我们会随机地将多少比例的隐藏层神经元的输出置为 0。

第 13 行：在第 2 层输出值中随机地将 `keep_prob` 比例的输出值保留，其余值变为 0，

形成新的输出值 `a_2_dropout`。

第 15 行：定义第 2 层（隐藏层）到第 3 层（输出层）的连接权值矩阵，用 0.0 来进行初始化。同样，为了计算方便，实际上定义的也是理论部分连接权值矩阵的转置 $(W^{(2)})^T$ 。

第 16 行：初始化第 3 层（输出层）的偏置值，用 0.0 进行初始化。

第 17 行：求出第 3 层（输出层）的输入值。

第 18 行：求出经过 **Softmax** 激活函数后的输出值，此值即每个类别的概率。

第 20、21 行：定义交叉熵。

第 22 行：采用随机梯度下降算法，学习率为 0.3。

第 23、24 行：代价函数为交叉熵函数再加上 L2 调整项（权值衰减）。

第 25~27 行：采用 Adam 优化算法。

第 28 行：利用 TensorFlow 的 `argmax` 函数，分别求出计算类别向量每个样本的最大值下标和类别向量每个样本的最大值下标，并对其进行比较。

第 29 行：求出预测精度，首先调用 TensorFlow 的 `cast` 函数，将第 16 行的结果变为浮点数列表，形式为：[1.0, 1.0, 0.0, 1.0, 1.0]，这里假设只有 5 个样本。再调用 TensorFlow 的 `reduce_mean` 函数求出这个列表的平均值：(1.0+1.0+0.0+1.0+1.0)/5=0.8，这个值就是模型预测的精度。

第 30、31 行：将模型中的变量和张量保存为本类的属性。

第 32、33 行：返回模型定义的 `X`, `y_`, `y`, `keep_prob`, `cross_entropy`, `train_step`, `correct_prediction`, `accuracy`。

第 35~52 行：将模型中的变量和张量保存为本类的属性的具体实现过程。

在具体查看训练方法之前，我们先来看一下基于 ReLU 神经元的训练效果，后台打印信息如图 4.11 所示。

```
4:400# train:0.9656909108161926, validation:0.9639999866485596
4:450# train:0.9629636406898499, validation:0.9613999724388123
4:500# train:0.965254545211792, validation:0.9660000205039978
5:0# train:0.966381847858429, validation:0.9674000144004822
5:50# train:0.9699272513389587, validation:0.9692000150680542
5:100# train:0.9673091173171997, validation:0.9682000279426575
5:150# train:0.966981828212738, validation:0.9652000069618225
5:200# train:0.9663636088371277, validation:0.9648000001907349
5:250# train:0.9677454829216003, validation:0.9674000144004822
5:300# train:0.9672726988792419, validation:0.9675999879837036
5:350# train:0.9681817889213562, validation:0.9674000144004822
5:400# train:0.9688727259635925, validation:0.9664000272750854
5:450# train:0.9654181599617004, validation:0.9624000191688538
5:500# train:0.9660000205039978, validation:0.9679999947547913
6:0# train:0.9677272439002991, validation:0.9656000137329102
6:50# train:0.9679818153381348, validation:0.9696000218391418
6:100# train:0.9667272567749023, validation:0.9660000205039978
6:150# train:0.9644545316696167, validation:0.9646000266075134
6:200# train:0.9685636162757874, validation:0.967199981212616
6:250# train:0.9682363867759705, validation:0.9696000218391418
6:300# train:0.9677090644836426, validation:0.9649999737739563
6:350# train:0.9668545722961426, validation:0.9661999940872192
6:400# train:0.967199981212616, validation:0.9688000082969666
6:450# train:0.9698181748390198, validation:0.9696000218391418
0.9674
```

图 4.11 ReLU 神经元训练结果

由上图可以看出，采用 ReLU 神经元时，模型在训练样本集上的识别精度为 96.98%，

在验证样本集上的识别精度为 96.96%，在测试样本集上的识别精度为 96.74%。而我们采用 Sigmoid 神经元时，在训练样本集上的识别精度为 97.6%，在验证样本集上的识别精度为 93.1%，在测试样本集上的精度为 92.8%。从这些数据可以看出，虽然 Sigmoid 神经元在训练样本集上识别精度略高一些，但是在验证样本集和测试样本集上的识别精度，都明显低于 ReLU 神经元，而且已经出现较为明显的过拟合趋势，而相对来讲，ReLU 神经元还处在欠拟合状态，我们可以通过改变 Early Stopping 标准，进一步提高其识别精度，而不用担心过拟合问题。事实上，根据我们的实践，通过调整模型的超参数，ReLU 神经元在测试样本集上的精度可以达到 98% 多一点，读者可以尝试调整模型的超参数，达到甚至超过这一指标。

我们再来看基于 ReLU 神经元的模型在训练样本集上、验证样本集上识别精度的变化趋势，如图 4.12 所示。

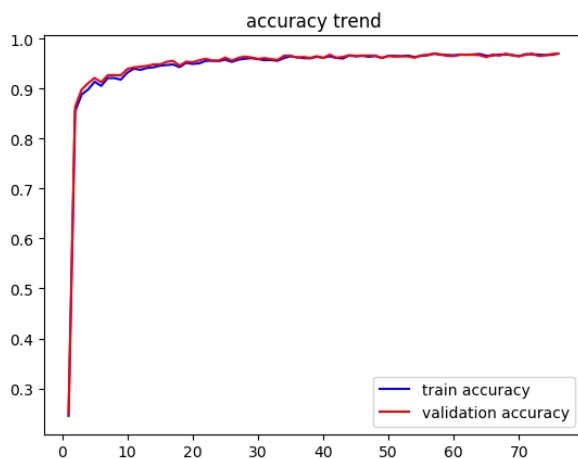


图 4.12 ReLU 神经元识别精度变化趋势

由上图可以看出，在训练样本集上的识别精度和在验证样本集上的识别精度曲线重合度很高，表明模型还有很大的提升空间，通过调整 Early Stopping 标准，我们可以进一步提高识别精度，读者可以自己进行这方面的实验，毕竟在实际工作中，对模型超参数的调整是一项非常重要的工作，这不仅需要我们有坚实的理论知识作为指导，还要求我们有丰富的调参实践经验。

下面我们来看模型的训练方法，代码如下：

```
1 def train(self, mode=TRAIN_MODE_NEW, ckpt_file='work/lgr.ckpt'):
2     X_train, y_train, X_validation, y_validation, X_test, \
3         y_test, mnist = self.load_datasets()
4     X, y_, y, keep_prob, cross_entropy, train_step, correct_prediction, \
5         accuracy = self.build_model()
6     epochs = 10
7     saver = tf.train.Saver()
8     total_batch = int(mnist.train.num_examples/self.batch_size)
9     check_interval = 50
10    best_accuracy = -0.01
11    improve_threthold = 1.005
```

```

12 no_improve_steps = 0
13 max_no_improve_steps = 3000
14 is_early_stop = False
15 eval_runs = 0
16 eval_times = []
17 train_accs = []
18 validation_accs = []
19 with tf.Session() as sess:
20     sess.run(tf.global_variables_initializer())
21     if Mlp_Engine.TRAIN_MODE_CONTINUE == mode:
22         saver.restore(sess, ckpt_file)
23     for epoch in range(epochs):
24         if is_early_stop:
25             break
26         for batch_idx in range(total_batch):
27             if no_improve_steps >= max_no_improve_steps:
28                 is_early_stop = True
29                 break
30             X_mb, y_mb = mnist.train.next_batch(self.batch_size)
31             sess.run(train_step, feed_dict={X: X_mb, y: y_mb,
32                 keep_prob: self.keep_prob})
33             no_improve_steps += 1
34             if batch_idx % check_interval == 0:
35                 eval_runs += 1
36                 eval_times.append(eval_runs)
37                 train_accuracy = sess.run(accuracy,
38                     feed_dict={X: X_train, y: y_train, keep_prob: 1.0})
39                 train_accs.append(train_accuracy)
40                 validation_accuracy = sess.run(accuracy,
41                     feed_dict={X: X_validation, y: y_validation,
42                         keep_prob: 1.0})
43                 validation_accs.append(validation_accuracy)
44                 if best_accuracy < validation_accuracy:
45                     if validation_accuracy / best_accuracy >= \
46                         improve_threthold:
47                         no_improve_steps = 0
48                         best_accuracy = validation_accuracy
49                         saver.save(sess, ckpt_file)
50                     print('{0}:{1}# train:{2}, validation:{3}'.format(
51                         epoch, batch_idx, train_accuracy,
52                         validation_accuracy))
53             print(sess.run(accuracy, feed_dict={X: X_test,
54                 y: y_test, keep_prob: 1.0}))
55         plt.figure(1)
56         plt.subplot(111)
57         plt.plot(eval_times, train_accs, 'b-', label='train accuracy')
58         plt.plot(eval_times, validation_accs, 'r-',
59             label='validation accuracy')
60         plt.title('accuracy trend')
61         plt.legend(loc='lower right')
62         plt.show()

```

第 2、3 行：读入训练样本集输入信号集、训练样本集标签集、验证样本集输入信号集、验证样本集标签集、测试样本集输入信号集、测试样本集标签集。

第 4、5 行：创建模型，这里创建的模型是前面讲到的 ReLU 神经元的模型。

第 6 行：学习整个训练样本集的遍数。

第 7 行：初始化 TensorFlow 模型保存和恢复对象 `saver`。

第 8 行：用用户训练样本集中样本数除以迷你批次大小，得到迷你批次数量 `total_batch`。

第 9 行：每隔 50 次迷你批次学习，计算在验证样本集上的精度。

第 10 行：保存在验证样本集上所取得的最好的验证样本集精度。

第 11 行：定义验证样本集上精度提高 0.5% 时才算显著提高。

第 12 行：记录在验证样本集上精度没有显著提高学习迷你批次的次数。

第 13 行：在验证样本集精度最大没有显著提高的情况下，允许学习迷你批次的次数。

第 14 行：是否终止学习过程。

第 15 行：评估验证样本集上识别精度的次数。

第 16 行：用 `eval_times` 列表来保存评估次数，作为后面绘制识别精度趋势图的横坐标。

第 17 行：用 `train_accs` 列表保存在训练样本集上每次评估时的识别精度，作为后面图形深色曲线的纵坐标。

第 18 行：用 `validation_accs` 列表保存在验证样本集上每次评估时的识别精度，作为后面图形浅色曲线的纵坐标。

第 19 行：启动 TensorFlow 会话。

第 20 行：初始化全局参数。

第 21、22 行：如果模式为 `TRAIN_MODE_CONTINUE`，则读入以前保存的 `ckpt` 模型文件，初始化模型参数。

第 23 行：循环第 20~42 行操作，对整个训练样本集进行一次学习。

第 24、25 行：如果 `is_early_stop` 为真，则终止本层循环。

第 26 行：循环第 23~42 行操作，对一个迷你批次进行学习。

第 27~29 行：如果验证样本集上识别精度没有显著改善的迷你批次学习次数大于最大允许的验证样本集上识别精度没有显著改善的迷你批次学习次数，则将 `is_early_stop` 置为真，并退出本层循环。这会直接触发第 24、25 行终止外层循环，学习过程结束。

第 30 行：从训练样本集中取出一个迷你批次的输入信号集 `X_mb` 和标签集 `y_mb`。

第 31、32 行：调用 TensorFlow 计算模型输出、代价函数，求出代价函数对参数的导数，并应用梯度下降算法更新模型参数值。注意，此时我们采用 Dropout 调整技术，将隐藏层神经元保留比例作为一个参数 `keep_prob` 传给模型。

第 33 行：将验证样本集没有显著改善的迷你批次学习次数加 1。

第 34 行：如果连续进行了指定次数的迷你批次学习，则计算统计信息。

第 35 行：识别精度评估次数加 1。

第 36 行：将识别精度评估次数加入 `eval_times`（图形横坐标）列表中。

第 37、38 行：计算训练样本集上的识别精度。

第 39 行：将训练样本集上的识别精度加入训练样本集识别精度列表 `train_accs` 中。

第 40~42 行：计算验证样本集上的识别精度。注意，此时采用 Dropout 调整技术，将隐藏层神经元保留比例作为一个参数 `keep_prob` 传给模型，这里给出的是 1.0，即全部保留，

这是一个惯例，即在训练时使用 Dropout 调整技术，在评估和运行时不使用 Dropout 调整技术，读者一定要注意。

第 43 行：将验证样本集上的识别精度加入验证样本集识别精度列表 `validation_accs` 中。

第 44 行：如果验证样本集上最佳识别精度小于当前的验证样本集上的识别精度，执行第 45~49 行操作。

第 45~47 行：如果当前验证样本集上的识别精度比之前的最佳识别精度提高 0.5% 以上，则将验证样本集没有显著改善的迷你批次学习次数设为 0。

第 48 行：将验证样本集上最佳识别精度的值设置为当前验证样本集上的识别精度值。

第 49 行：将当前模型参数保存到 `ckpt` 模型文件中。

第 50~52 行：打印训练状态信息。

第 53、54 行：训练完成后，计算测试样本集上的识别精度，并打印出来。

第 55、56 行：初始化 `matplotlib` 绘图库。

第 57 行：绘制训练样本集上识别精度的变化趋势曲线，用蓝色绘制。

第 58、59 行：绘制验证样本集上识别精度的变化趋势曲线，用红色绘制。

第 60 行：设置图形标题。

第 61 行：在右下角添加图例。

第 62 行：具体绘制图像。

运行这个程序，我们就可以得到前文中 Sigmoid 及 ReLU 神经元模型的后台输出和识别精度曲线图了。

我们的模型在训练完成之后，就会进入运行阶段，我们这里会给出两个实例，一个是任意从测试样本集中取出一个样本，让我们的模型进行预测，在上一章利用逻辑回归进行手写数字识别时，对于印刷体数字 5 组成的图片，虽然对我们来说识别起来很简单，但是逻辑回归算法却会出现识别错误，将其识别为 6，我们还以此为例，利用多层感知器模型，重新进行识别，来比较一下这两个模型在这个特定应用场景下的效果。

下面我们来看 `run` 方法，代码如下：

```
1 def run(self, ckpt_file='work/lgr.ckpt'):
2     img_file = 'datasets/test6.png'
3     img = io.imread(img_file, as_grey=True)
4     raw = [1 if x<0.5 else 0 for x in img.reshape(784)]
5     sample = np.array(raw)
6     X_train, y_train, X_validation, y_validation, \
7         X_test, y_test, mnist = self.load_datasets()
8     X, y_, y, keep_prob, cross_entropy, train_step, correct_prediction, \
9         accuracy = self.build_model()
10    #sample = X_test[102]
11    X_run = sample.reshape(1, 784)
12    saver = tf.train.Saver()
13    digit = -1
14    with tf.Session() as sess:
15        sess.run(tf.global_variables_initializer())
16        saver.restore(sess, ckpt_file)
17        rst = sess.run(y_, feed_dict={X: X_run, keep_prob: 1.0})
18        print('rst:{0}'.format(rst))
19        max_prob = -0.1
```

```

20     for idx in range(10):
21         if max_prob < rst[0][idx]:
22             max_prob = rst[0][idx]
23             digit = idx
24     # W_1_1
25     W_1 = sess.run(self.model['W_1'])
26     wight_map = W_1[:,0].reshape(28, 28)
27     a_2 = sess.run(self.model['a_2'], feed_dict={X: X_run, \
28             keep_prob: 1.0})
29     a_2_raw = a_2[0]
30     a_2_img = a_2_raw[0:484]
31     feature_map = a_2_img.reshape(22, 22)
32     img_in = sample.reshape(28, 28)
33     plt.figure(1)
34     plt.subplot(131)
35     plt.imshow(img_in, cmap='gray')
36     plt.title('result:{0}'.format(digit))
37     plt.axis('off')
38     plt.subplot(132)
39     plt.imshow(wight_map, cmap='gray')
40     plt.axis('off')
41     plt.title('wight row')
42     plt.subplot(133)
43     plt.imshow(feature_map, cmap='gray')
44     plt.axis('off')
45     plt.title('hidden layer')
46     plt.show()

```

第 2 行：用绘图软件做一个 28×28 的图像，在上面写一个数字，这里我们直接写上一个印刷体的 5。

第 3 行：以灰度图像方式读出图像内容。

第 4 行：首先将其形状从 28×28 二维变为一维 784，然后根据每个像素的值进行处理：值小于 0.5 时取 1，否则取 0。

第 5 行：将其变为 numpy 数组，作为一个样本。

第 6、7 行：调用 load_datasets 方法，读入训练样本集、验证样本集、测试样本集的内容。

第 8、9 行：调用 build_model 方法，建立逻辑回归模型。

第 10 行：取测试样本集中的第 103 个样本作为测试样本，我们的模型在训练过程中没有见过测试样本集中的样本，因此可以模拟实际应用中遇到的情况。

第 11 行：将其变为[1, 784]的矩阵形式，我们可以称之为运行样本集，其中只有一个样本。

第 12 行：初始化 TensorFlow 的 saver 对象。

第 13 行：定义 digit 为最终识别出的 0~9 的数字，取-1 表示还没有识别结果。

第 14 行：启动 TensorFlow 会话来运行程序。

第 15 行：初始化变量。

第 16 行：恢复之前保存的模型参数文件，并初始化模型参数。

第 17 行：求以样本 X_run 为输入，在输出层经过 Softmax 函数后的计算值，表示为每个类别的概率。注意：这里设置 Dropout 技术中隐藏层神经元的保留率为 1.0，即所有隐藏层神经元均参与运算，相当于多个随机网络共同投票得出最终结果。

第 18 行：打印出所有类别的概率值。

第 19 行：定义 `max_prob` 记录所有类别最大的概率值。

第 20 行：循环识别结果 `rst` 每个类别的概率。

第 21~23 行：如果最大概率小于当前类别的概率，将当前概率赋给最大概率，识别出的数字等于类别索引值，即对应的 0~9 中的数字。当循环完所有类别后，我们就能找到概率最大的类别及其所对应的数字了。

第 25 行：取出输入层到隐藏层的连接权值矩阵 `W_1`（实际为其转置）。

第 26 行：将其第一行的形状转为 28×28 的图片数据格式。

第 27、28 行：求出隐藏层神经元输出值。注意：这里设置 Dropout 技术中隐藏层神经元的保留率为 1.0，即所有隐藏层神经元均参与运算，相当于多个随机网络共同投票得出最终结果。

第 29 行：取出隐藏层输出的原始数据。

第 30 行：由于隐藏层输出数据为 512 维，但我们想显示成一个正方形数据，所以只取前面的 484（22×22）维数据。

第 31 行：将隐藏层输出前 484 维变为 22×22 的特征图。

第 33、34 行：初始化 `matplotlib` 绘图库。

第 35~37 行：绘制识别的输入图像，将识别结果显示在标题上。

第 38~41 行：显示输入层到隐藏层第一行连接权值的图像。

第 42~45 行：显示隐藏层输出值图像，我们知道，多层感知器是将原始输入信号进行变换，变为适合分类识别的形式，可以视为一种特征学习形式。

第 46 行：同时显示三幅图像。

我们先来看对测试样本集中的第 103 个图片的识别过程，后台输出结果如图 4.13 所示。

```
rst:[[ 1.53329202e-05  2.15204309e-07  6.32943681e-07  2.46642786e-03
 5.28531075e-07  9.96973753e-01  2.41150985e-07  2.76966166e-04
 5.86624592e-05  2.07241930e-04]]
```

图 4.13 识别测试样本集中的图片

如上图所示，显示了 0~9 个数字全部 10 个类别对应的概率值，最大值出现在第 6 项，即对应数字 5 的位置，而且与其他类别相比，概率差别非常大，表明模型非常确定这张图片是数字 5。

识别图片、权值向量、特征图如图 4.14 所示。



图 4.14 输入图片、连接权值和隐藏层特征图

还记得我们在逻辑回归模型中最后测试的例子吗？我们通过绘图软件，生成一个黑底白字印刷体的数字 3，虽然我们可以很清楚地看出是数字 5，但是逻辑回归模型却无法正确识别，我们当时的解释是因逻辑回归模型只依赖于分类函数的局部光滑性，对扰动比较敏感所致。实际上，逻辑回归模型只能解决线性可分问题，而图像分类问题通常不是线性可分的，所以逻辑回归模型不可能有特别完美的识别结果。

下面我们将这个例子输入当前的多层感知器模型中，看看会出现什么结果，运行结果如图 4.15 所示。

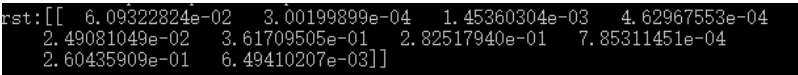


图 4.15 生成图片识别

如上图所示，显示了 0~9 个数字全部 10 个类别对应的概率值，最大值出现在第 6 项，即对应数字 5 的位置，但是与上例不同，在第 9 个位置，对应数字 8 的概率，与这个值相差很小，这表明模型虽然得到了正确的结果，但是置信度并不高。无论如何，我们的多层感知器模型还是可以成功地进行识别的。之所以逻辑回归模型无法正确识别，而多层感知器模型可以正确识别，最主要的原因就是多层感知器模型引入了中间的隐藏层，隐藏层通过特征学习，将原来线性不可分的问题，转化为在新的特征空间中线性可分的问题了，从而使模型可以处理线性不可分问题。

输入图片、权值向量、特征图如图 4.16 所示。

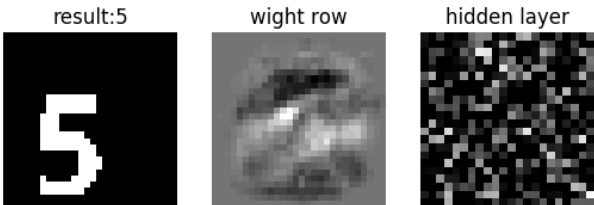


图 4.16 输入图片、连接权值和隐藏层特征图

在本章的例子中，我们并没有着力去优化模型的超参数，所以在测试样本集上的识别精度也没有达到当前的最好水平。根据 3.3 节关于 MNIST 手写数字识别数据集的介绍，我们可以看到多层感知器模型当前可以达到的最高识别精度如表 4.1 所示。

表 4.1 MNIST数据各算法的最佳表现

类 型	分 类 器	变形处理	预 处 理	误差 (%)
神经网络1	2层: 784-800-10	否	否	1.6
神经网络2	2层: 784-800-10	是	否	0.7
深层网络	6层: 784-2500-2000-1500- 1000-500-10	否	否	0.35

读者可以按照神经网络 1 和深层网络的网络架构超参数，训练自己的多层感知器模型，尝试达到上表中的结果，当然这就需要读者综合运用 L2 调整项（权值衰减）、Early Stopping 技术、Dropout 技术，还要合理选择优化算法和学习率等超参数。这个过程需要反复测试，比较麻烦，但是这是深度学习在实际应用中重要的一环，在这个过程中积累的经验对实际的工程项目非常有用。

第 5 章

卷积神经网络

本章将讨论卷积神经网络，这是最早应用于实践的深度学习网络，也是最成功的深度学习网络，在图像识别领域是当之无愧的王者。在 MNIST 手写数字识别竞赛中，在测试样本集上取得了 0.21% 的误差率（由 35 个卷积神经网络投票产生的结果），是目前最佳的识别效果，优于其他对比模型，也高于人类的识别准确率。在自然景物识别的 ImageNet 竞赛中，以卷积神经网络为基础的 GoogleNet，也取得了一次识别准确率达 68% 的成绩，取得了 2014 年的冠军。目前，在主流图像识别应用系统中，使用的基本都是卷积神经网络。

本章首先介绍对卷积神经网络的直观理解，然后讲述卷积神经网络的数学基础，再讨论卷积神经网络的迁移学习和微调技术，即使用别人已经训练好的网络来处理新问题，也不用从头开始训练网络。最后介绍卷积神经网络在 MNIST 手写数字识别中的应用。

5.1 卷积神经网络原理

5.1.1 卷积神经网络的直观理解

在讨论卷积神经网络之前，我们先根据常识来讨论一下怎样可以提高图像识别的准确率。以印刷字母识别为例，假设需要网络识别大写字母 A，我们给网络的训练样本可能如图 5.1 所示。



图 5.1 印刷体字母识别

但是在实际应用中，样本中字母 A 的位置可能发生偏移，形成如图 5.2 所示的情况。



图 5.2 字母位置变化

同时，不仅字母的位置可能发生变化，字母的大小也可能变化，如图 5.3 所示。

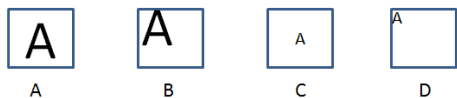


图 5.3 字母大小、位置的变化

在多数情况下还会发生角度变化，如图 5.4 所示。

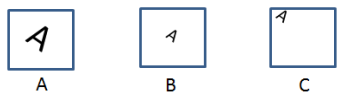


图 5.4 字母的大小、位置、旋转角度的变化

上述情况都会对字母识别问题造成影响。这里只列举了印刷体字母的识别，手写体的字母还有笔迹的问题等。另外，对于真实景物而言，还会有光照、纹理、遮挡等问题，使模式识别任务变得更加复杂。

在实际的图像识别领域，以上各种方式通常是以组合形式出现的，即图像中的元素需要经过一系列的平移、旋转、缩放后，才能得到与训练样本相似的标准图像，因此在传统的图像识别中，需要对图像进行预处理。在神经网络进行图像识别中，我们也希望神经网络可以自动处理这些变换，用学术术语来讲，就是具有平移、旋转、缩放的不变性，卷积神经网络就是为解决这一问题而提出的一种架构。需要注意的是，如果只采用卷积神经网络，我们只能获得一定程度的平移不变性，对旋转和缩放不变性，基本无法取得任何效果。因此在实际应用中，我们通常将空间变换网络（Spatial Transformer Network）和卷积神经网络结合起来使用，这样可以较好地解决图像的旋转、平移、缩放的不变性问题，取得较好的图像识别效果。这部分内容不是本书的重点，有兴趣的读者可以参考下面的论文：<https://arxiv.org/abs/1506.02025>。

那么怎样才能让神经网络具有我们希望的这种变换不变性呢？我们知道，神经网络的兴起，很大程度上是仿生学在人工智能领域的应用，我们用人工神经元模型及其连接来模仿人类大脑，解决一些常规方法不能解决的复杂问题。对于图像识别而言，神经网络的研究人员，也希望通过模拟大脑视觉皮层的处理机制，来提高图像识别的准确率。

Hubel and Wiesel 对猫的视觉皮层的研究表明，视觉皮层细胞会组成视觉接收域，只负责对一部分图像信号的处理，处理局部的空间信息，例如图像的边缘识别等。同时，视觉皮层中存在两类细胞，一类细胞是简单细胞，主要用于识别图像边缘等基本信息；一类是复杂细胞，具有位置不变性，可以识别各种高级的图像信息。

以上述发现为指导，研究人员提出了卷积神经网络模型，其主要包括两大方面特性：层间稀疏连接和共享连接权值。

层间稀疏连接主要模拟大脑视觉皮层的接收域，以具有简单细胞和复杂细胞两类不同细胞，分别处理局部细节和全局空间不变性。首先将图像像素分为 3×3 的区域，所以对于 $l=1$ 的输入层而言，这 9 个像素连接到 9 个输入层神经元，而这 9 个神经元只连接到 $l=2$ 上的一个神经元，如图 5.5 所示。注意由于我们画的是二维图，因此只显示面对我们的三个神经元。

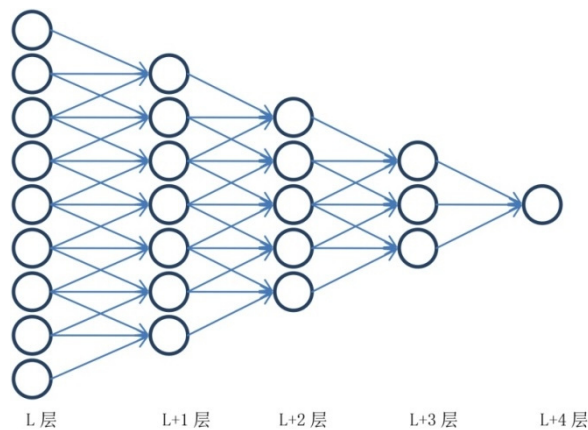


图 5.5 层间稀疏连接

如图所示，每一层都只与其底层 3×3 的神经元相连接，这样最上层神经元对应的视觉接收域将变为 9×9 。利用上述结构，可以采用多层来表示原来的图像信息，底层神经元主要负责边缘等基本信息的识别，越往高层走识别的级别越高，最上层则可以表达为我们希望区分的类别。这其实与传统的数字图像处理中金字塔模型比较类似，解决的是同一类问题。

共享连接权值如图 5.6 所示。

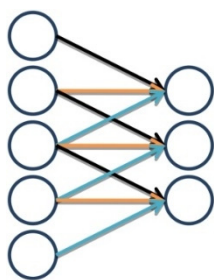


图 5.6 共享连接权值

图中不同颜色的连接具有相同的连接权值。因此，考虑到这种情况，在卷积神经网络实现中，需要对原来的算法进行修改，将由对单个权值求导变为对三个权值之和进行求导。

通过共享权值模式，可以使卷积神经网络识别出图像中的物体，而与物体的空间位置无关，即实现本文开头所提到的旋转、平移、缩放的不变性，这对于在图像识别领域经常出现的物体在图像中的位置变化、大小变化、观察角度变化所造成的识别困难，具有非常

好的解决效果。同时，权值共享减少了网络的参数个数，也大大提高了网络的学习效率，因此卷积神经网络成为图像识别一个事实上的标准。

5.1.2 卷积神经网络构成

如果直接将多层感知器模型应用于 CIFAR-10 的图像识别应用中，我们知道 CIFAR-10 是 32×32 的彩色图像，因此输入层就需要 3072 ($32 \times 32 \times 3$) 个神经元（因为彩色图像每个像素点由 R、G、B 三色组成）。假设第一个隐藏层的数目为输入层的一半，则第一个隐藏层的神经元数量为 1536，采用全连接形式，连接权值为 $(3072+1) \times 1536 + 1536 = 4721664$ ，即共有 400 多万个连接权值。即使网络只有三层，输出层为 10 个类别，则输出层的连接权值为 $(1536+1) \times 10 + 10 = 15380$ ，所以总的网络参数为 480 万个左右，而这仅是一个非常简单的图像识别应用。对于更复杂的 ImageNet 图像识别任务，由于其都是高分辨率图像，实际中一般将其向下采样 256×256 个图片，如果 ImageNet 数据集采用多层感知器模型，那么输入层神经元数为 65536 (256×256) 个，隐藏层取输入层的 50%，则有 32768 个神经元，则需要的参数为 $2147549184[(65536+1) \times 32768 + 32768]$ 个，仅这一层就会有 21 亿个参数，显然这么多参数要经过梯度下降算法学习得到，将是一个极其漫长的过程，在实际应用中，基本不具备可行性。因此多层感知器模型中的全连接性，不仅造成很大的资源浪费，还会出现过拟合问题。

卷积神经网络考虑到输入信号是图像的特点，参考生物视觉神经的接收域概念，采用了稀疏连接、权值共享、池化等概念，成功解决了全连接模型在图像处理领域的不足，取得了相当大的成功。

卷积神经网络的层由三个维度组成：宽度、高度和深度。对于彩色图像而言，例如 10×10 图像，由于彩色图像每个像素是由 R、G、B 三原色组成，因此这个图像可以表示为红色 R 分量的宽度 \times 高度 $= 10 \times 10$ 矩阵，紧接着是绿色 G 分量宽度 \times 高度 $= 10 \times 10$ 矩阵，最后是蓝色 B 分量宽度 \times 高度 $= 10 \times 10$ 矩阵。与此对应，卷积神经网络的输入层即 $10 \times 10 \times 3$ 个神经元的输入层，按照 R 分量矩阵第一行 10 个像素，第 2 行 10 个像素，直到 R 分量第 10 行的 10 个像素；接着是 G 分量矩阵第一行 10 个像素，第 2 行 10 个像素，直到 G 分量矩阵的第 10 个分量；最后是 B 分量矩阵第一行 10 个像素，第 2 行 10 个像素，直到 B 分量矩阵的 10 个像素，如图 5.7 所示。

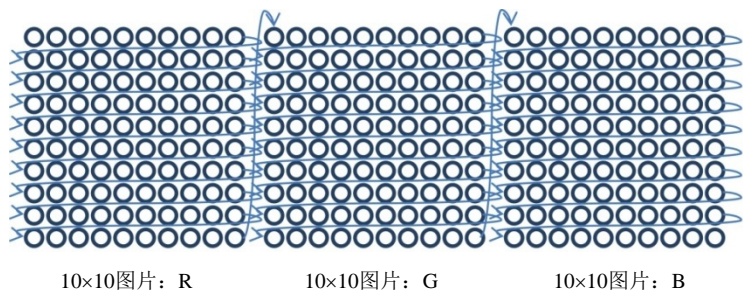


图 5.7 层的定义方式

假设图像的宽度为 w ，高度为 h ，R 分量矩阵像素点表示为 $(0,i,j)$ ，G 分量矩阵像素点表示为 $(1,i,j)$ ，B 分量矩阵像素点表示为 $(2,i,j)$ 。如果想知道 G 分量矩阵的第 i 行第 j 列像素点对应的输入层神经元，其公式为： $1 \times 10 \times 10 + i \times 10 + (j+1)$ ，其中 i 和 j 以 0 开始，输入层神经元编号从 1 开始。同理，如果知道输入层神经元编号，如 289，也可以计算出其对应哪个分量矩阵的哪一行哪一列，由 $289 / 100 = 2$ 可知其在 B 分量矩阵，由 $89 / 10 = 8$ 可知其在第 $(8+1)$ 行，由 $9 / 10 = 8$ 可知其在第 $8+1$ 列。

因此如果对 CIFAR-10 这样的图像数据集应用卷积神经网络，输入层就可以定义为 $32 \times 32 \times 3$ 。而 ImageNet 图像如果规整为 256×256 时，输入层就可以定义为 $32 \times 32 \times 3$ 。

与多层感知器由输入层、隐藏层和输出层组成不同，卷积神经网络由输入层、卷积层、ReLU 层、池化层、全连接层、输出层组成。

典型的卷积神经网络架构如图 5.8 所示。



图 5.8 典型的卷积神经网络架构

输入层：输入层每个神经元对应图像中的像素点，例如对于 CIFAR-10 图像识别数据集，分辨率为 32×32 的彩色图像，其表示形式为 $32 \times 32 \times 3$ 。

卷积层：卷积层神经元与前一层对应分量矩阵中的接收域内的神经元相连接，将连接权值与接收域内神经元输出做点积，然后经过本神经元的激活函数，生成本神经元的输出。卷积层还有一个重要的参数——滤波器的数量。

ReLU 层：该层神经元的激活函数为 $g(x) = \max\{0, x\}$ ，即输入值小于 0 时为 X 轴，大于 0 时为 $y=x$ 直线。

池化层：执行类似图像向下采样操作，目前采用最多的是最大池化，也有一些系统采用平均池化，如果输入信号为 $32 \times 32 \times d$ ，其中 d 为深度，则进行 2×2 最大池化之后变为 $16 \times 16 \times n$ ，并传播给下一层。

全连接层：这一层与多层感知器模型隐藏层加输出层的结构类似，如果是对 CIFAR-10 数据集进行图像识别，则最后一层为 $1 \times 1 \times 10$ ，对应于该数据集的 10 个分类，输出值代表每个类别的概率。

由此可见，卷积神经网络是由原始图像像素值经过一系列卷积层、ReLU 层、池化层，最后经全连接层，转化为图像类别的出现概率。在卷积层和全连接层的输出值由连接权值、偏移量和激活函数共同确定，根据梯度下降算法调整连接权值和偏移量。而 ReLU 层和池化层没有参数，只进行截断输出或向下采样操作，因此不需要学习。

1. 卷积层

卷积层是卷积神经网络最重要的部分，完成绝大部分计算，是卷积网络能够成功应用于图像识别领域的关键。

在具体讲述卷积层操作之前，我们先来复习一下数学上的卷积操作。正是由于数学上的卷积操作，才使得我们可以得到图像的大小、位置、角度的不变性。

对于一维信号，卷积定义为：

$$O(x) = f(x) \times g(x) = \sum_{u=-\infty}^{\infty} f(u)g(x-u) = \sum_{u=-\infty}^{\infty} f(x-u)g(u)$$

而我们要处理的图像信号是二维信号，卷积定义为：

$$f(m,n) \times g(m,n) = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} f(m,n)g(m-u,n-v) = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} f(m-u,n-v)g(m,n)$$

在图像处理中，我们知道，对图像进行卷积操作只要选择合适的核（Kernel），即可取得图像中的边缘。

我们先来看卷积层与上一层的连接形式，以输入层与卷积层的连接为例，池化层与卷积层的连接与此类似。

假设接收域为5×5，则卷积层第一个神经元与输入层连接的神经元如图5.9中涂色的5×5神经元块右下角3×3的神经元所示，共有75（5×5×3）个连接权值，其本身还有1个偏移量，所以共有76个参数。

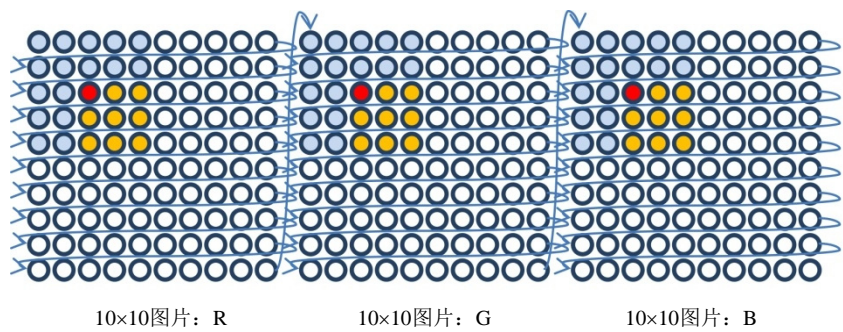


图 5.9 卷积层连接形式

在信号前向传播阶段，首先进行卷积操作，即将连接权值与对应输入层神经元输出相乘并求其和，将求出的和加上该神经元偏移量，再输入到激活函数中，求出激活函数的值作为本神经元的输出。

卷积层也是有深度的，以上讨论是针对卷积层深度上的第一层来讨论的，假设卷积层的深度为9，在其深度的每一层上，第一个神经元都会与图5.9中涂色的5×5神经元块右下角3×3的神经元相连接，也都有76个参数，可以用同样的方法做卷积，并求出该神经元的输出。

卷积层深度的每一层也叫滤波器，可以将其称为滤波器数量，以下对于深度的每一层称为第几个滤波器。例如我们要研究的是第3行第2列卷积层第5个滤波器，就是卷积层第5层上第3行第2列的神经元。

需要注意的是，对于边缘神经元，例如第 0 行第 0 列的卷积层神经元，我们会采用边缘填充 0 的方式，得到其完整的接收域，图 5.9 中涂色 5×5 神经元左侧及顶部两列和两行的神经元就是填充的神经元。

正如前文我们所讨论的，基于层间全连接的多层感知器模型不能很好地适应图像识别领域，卷积神经网络通过层间稀疏连接解决了这个问题。每个卷积层神经元，只与其接收域内的输入层神经元相连接，但是需要注意，这里所指的层间稀疏连接仅指宽度、高度方向，深度方向依然是全连接的。以 CIFAR-10 图像识别数据集为例，其输入层为 $32 \times 32 \times 3$ 个神经元，我们考察卷积层第 1 个滤波器的第 3 行第 5 列的神经元，其会和输入层 R 分量矩阵左上角坐标为(1,3)、右下角坐标为(5,7)的子矩形全连接，同时与输入层 G 分量矩阵左上角坐标为(1,3)、右下角坐标为(5,7)的子矩形全连接，以及与输入层 B 分量矩阵左上角坐标为(1,3)、右下角坐标为(5,7)的子矩形全连接。由此可见，卷积层神经元只与 R、G、B 分量矩阵上一个 5×5 的接收域进行全连接，即其与输入层连接在宽度、高度方向具有稀疏性，但是其同时与 R、G、B 分量都连接，即在深度方向上不具有稀疏性。

到目前为止，我们只讨论了卷积层神经元与输入层神经元稀疏连接方式，并且讨论了卷积层的一个超参数（不能通过学习算法改变的参数）滤波器数量，该值决定卷积层的深度。下面我们来看卷积层神经元数量的确定。

卷积层神经元的数量由三个因素决定：深度（滤波器数量）、步长、0 填充数量。

我们首先来讨论卷积层深度，也就是卷积层滤波器数量。每个卷积层滤波器可以响应输入图像的一个特征，如水平边缘、竖直边缘、色块等，对输入层一个特定的接收域，会有一系列卷积层神经元与其相连接，我们称这些神经元为深度柱或纤维。例如，在 CIFAR-10 图像识别任务中，输入层为 $32 \times 32 \times 3$ ，接收域为 5×5 ，卷积层深度（滤波器数量）为 8，因为我们想识别水平边缘、竖直边缘、色块、纹理等 8 个要素，此时对于输入层任意一个接收域，在卷积层有 8 个神经元与其进行全连接。

步长就是在对输入层做卷积时的步长。如果步长为 1，则对输入层每个节点均做卷积运算，并将值赋给卷积层神经元，因此卷积层宽度和高度与输入层宽度和高度相同（但是卷积层深度通常与输入层不同，输入层如果是原始图像的话，输入层深度为 3，而卷积层深度由超参数决定）。

因为要对输入层做卷积操作，但是对于边缘节点，我们无法做卷积操作，因此通常会在边缘填充 0，这样便于做卷积操作，同时不影响卷积的结果。例如在前面的例子中，图 5.9 中涂色 5×5 神经元左侧及顶部两列和两行的神经元就是 0 填充的神经元，此时 0 填充的数量为 2。

如果给定输入层宽度 W ，在这里我们假设图像宽度和高度相等，如果不相等，可以采用 0 填充方式变为宽度和高度相等。卷积层神经元接收域大小为 F ，在做卷积操作时的步长为 S ，边缘节点 0 填充数量为 P ，可以通过下面的公式计算出卷积层宽度（假设卷积层宽度=高度）：

$$w_{\text{conv}} = \frac{w_{\text{in}} - F + 2P}{S} + 1 \quad (1)$$

我们先举一个一维卷积的例子，如图 5.10 所示。

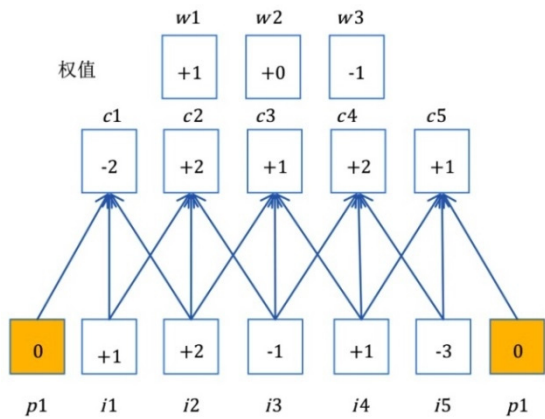


图 5.10 一维卷积层示例 $S=1$

在上图中，输入信号 $W=5$ ，为中间编号为 $i1$ 、 $i2$ 、 $i3$ 、 $i4$ 、 $i5$ 的神经元，为了便于进行卷积运算，进行了 0 填充，如图中左右两侧编号为 $p1$ 、 $p2$ 的神经元，因此 $P=1$ ，假设接收域为 $F=3$ ，做卷积时步长为 $S=1$ ，根据式 (1) 有：

$$w_{\text{conv}} = \frac{w_{\text{in}} - F + 2P}{S} + 1 = \frac{5 - 3 + 2 \times 1}{1} + 1 = 5$$

所以卷积层有 5 个神经元。根据卷积层与输入层之间的稀疏连接，接收域为 3，其连接形式如图 5.10 所示。图中神经元内数字为神经元输出值，外边的数字为神经元编号，在已经确定输入层神经元输出值的情况下，可以通过卷积操作求出卷积层输出值，例如在求卷积层第 1 个神经元 $c1$ 时，其公式为：

$$c1 = p1 \times w1 + i1 \times w2 + i2 \times w3 = 0 \times 1 + 1 \times 0 + 2 \times (-1) = -2$$

其余卷积层神经元值以此类推。

在上面的实例中，假设卷积时步长为 $S=1$ 。如果 $S=2$ ，其就会变为图 5.11 所示情形。

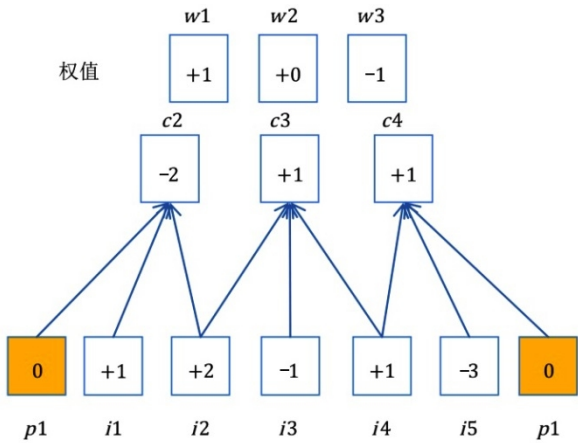


图 5.11 一维卷积层示例 $S=2$

在上图中，输入信号 $W=5$ ，为中间编号为 $i1$ 、 $i2$ 、 $i3$ 、 $i4$ 、 $i5$ 的神经元，为了便于进行卷积运算，进行了 0 填充，如图中左右两侧编号为 $p1$ 、 $p2$ 的神经元，因此 $P=0$ ，假设接收域为 $F=3$ ，做卷积时步长为 $S=2$ ，则：

$$w_{\text{conv}} = \frac{w_{\text{in}} - F + 2P}{S} + 1 = \frac{5 - 3 + 2 \times 1}{2} + 1 = 3$$

所以卷积层有 3 个神经元。根据卷积层与输入层之间的稀疏连接，接收域为 3，其连接形式如图所示。图中神经元内数字为神经元输出值，外边的数字为神经元编号，在已经确定输入层神经元输出值的情况下，可以通过卷积操作求出卷积层输出值，例如在求卷积层第 1 个神经元 $c1$ 时，其公式为：

$$c1 = p1 \times w1 + i1 \times w2 + i2 \times w3 = 0 \times 1 + 1 \times 0 + 2 \times (-1) = -2$$

其余卷积层神经元值以此类推。

以上实例为一维卷积情况，我们来看一个具体的实例，例如在 2012 年获得 ImageNet 竞赛第一名的网络，第一个卷积层的输入层为 $227 \times 227 \times 3$ ，接收域为 11，卷积操作步长为 4，没有采用 0 填充，则：

$$w_{\text{conv}} = \frac{w_{\text{in}} - F + 2P}{S} + 1 = \frac{227 - 11 + 2 \times 0}{4} + 1 = 55$$

卷积层深度 $K=96$ ，因此其卷积层尺寸为 290400 ($55 \times 55 \times 96$) 个神经元。

读者在这里可能会问，如果图像宽度和高度不相等，那么是否只有使它们一样才可以呢？其实宽度和高度不同也是可以进行卷积操作的，只是就需要区分宽度和高度了，公式如下：

$$w_{\text{conv}} = \frac{w_{\text{in}} - F + 2P_w}{S_w} + 1$$

$$h_{\text{conv}} = \frac{h_{\text{in}} - F + 2P_h}{S_h} + 1$$

式中， P_w 为宽度方向的 0 填充数量， P_h 为高度方向的 0 填充数量， S_w 为宽度方向步长， S_h 为高度方向卷积步长。但是这样进行处理增加了程序的复杂度，而且没有显著的好处，因此在实践中，基本都采用宽度和高度相等来进行处理。

讨论完层间稀疏连接之后，我们来讨论一下权值共享。卷积层通过层间稀疏连接，可以有效减少所需参数，再通过权值共享机制进一步减少参数。还是以 2012 年取得 ImageNet 竞赛冠军的网络为例，对于 $227 \times 227 \times 3$ 的彩色图像，如果采用 11×11 卷积，步长为 4，0 填充数量为 0，并且卷积层深度为 96，则卷积层神经元数量为 290400 ($55 \times 55 \times 96$)。而每个卷积层神经元与连接层 R、G、B 分量 11×11 子矩阵相连接，即共有 363 ($11 \times 11 \times 3$) 个连接权值，再加上其自身偏置值，共有 364 个参数，则输入层到卷积层的参数为 105705600 (290400×364) 个，即共有 1 亿多个参数。很显然，这也是一个非常大的数目。因此，仅有层间稀疏连接，对于高分辨率图像识别任务来说，还是远远不够的。

根据我们的讨论，卷积层是为了识别图像的特征，例如边缘、区域等，根据数字图像处理方面的知识，比如识别边缘的算子，也和卷积层神经元与输入层神经元之间的卷积操

作很类似，因此我们有理由相信，卷积层同一深度的滤波器或者说连接权值应该是相等的，因为它们都是处理同一特征的。也就是说，在卷积层深度为 1 时，共有 55×55 个卷积层神经元，每个神经元与输入层神经元有 $11 \times 11 \times 3$ 个连接权值，再加上自己的偏移量共有 364 个参数，而根据上面的分析，这 55×55 个卷积层神经元的 364 个参数全部是相同的。如果卷积层深度为 96，则只需有 34944 (96×364) 个参数即可。这就是卷积神经网络中的权值共享。

因为卷积层同一深度层上的神经元，与输入层神经元连接权值相等，所以我们求每个卷积层神经元与其所连接的输入层神经元的点积操作，就变成了卷积操作，这也是卷积网络得名的原因，而连接权值又被称为滤波器或核。

以上讨论是卷积神经网络在图像处理中进行的简化，但是对于某些具体问题，权值共享可能就不太适合了。例如我们的任务是人脸识别，假设人脸在图像中央，我们还知道某个特殊的滤波器，即 $11 \times 11 \times 3$ 的连接权值组合，对于识别人脸最有效，这时我们就会希望在图像中央区域卷积层神经元的连接权值取这个更有效的连接权值，因此在实际应用中，为了取得更好的效果，我们可能会放宽权值共享的限制。另外，对于不同类型的图像，如自然风光、人物照、商品照片等，可能需要侧重不同的特征，因此希望卷积层深度方向的滤波器可以根据图像类型动态调整，这样可以进一步提高图像识别效果。这就涉及对图像进行预处理，根据预处理结果动态组合出适合当前图像的卷积神经网络架构，从而取得更好的应用效果。关于这方面的内容，读者如果有兴趣，可以作进一步研究，也许可以取得一些突破性进展。

下面来总结一下卷积层的知识点。

- ☐ 输入层：宽度 $W1 \times$ 高度 $H1 \times$ 深度 $D1$ 。
- ☐ 卷积层。
- ☐ 超参数。
- ☐ 卷积层深度（滤波器数量） $K=D2$ 。
- ☐ 接收域宽度 F 。
- ☐ 卷积操作步长 S 。
- ☐ 边缘 0 填充 P 。
- ☐ 卷积层神经元数量：宽度 $W2 \times$ 高度 $H2 \times$ 深度 $D2$ 。

$$W2 = \frac{W1 - F + 2P}{S} + 1$$

$$H2 = \frac{H1 - F + 2P}{S} + 1$$

$$D2=K$$

- ☐ 权值共享：在卷积层深度方向第 d 个平面，共有 $F \times F$ 个卷积层神经元，其与输入层深度方向三个层 $F \times F$ 子区域做全连接，共有 $F \times F \times D1$ 个连接权值，卷积层深度为 K ，则卷积层参数为 $F \times F \times D1 \times K + F \times F \times K$ （卷积层神经元偏置值）。

卷积层神经元组织形式与输入层相同，在深度方向由 K 个切片平面组成，每个切片平面按先后列形式进行存储。

以上这些内容比较抽象，下面我们以一个具体的例子来说明这些概念。我们的输入图像是一个 5×5 的彩色图像，因此输入层为 5×5×3 个神经元，卷积核中元素下标排列规则为：在最左侧从上到下按 R、G、B 分量排列，用 $x[\text{行}, \text{列}, \text{深度切片 (RGB 分量)}]$ 表示输入层神经元，即其参数为 $W1=5, H1=5, D1=3$ 。

下面我们来确定卷积层超参数，卷积层滤波器数量为 $K=D2=2$ ，即深度为两层。接收域为 3×3，即 $F=3$ ，卷积操作步长 $S=2$ ，边缘 0 填充数量为 $P=1$ ，有了这些参数，就可以求出卷积层相关参数。

$$W2 = \frac{W1 - F + 2P}{S} + 1 = \frac{5 - 3 + 2 \times 1}{2} + 1 = 3$$
$$H2 = \frac{H1 - F + 2P}{S} + 1 = \frac{5 - 3 + 2 \times 1}{2} + 1 = 3$$
$$D2 = K = 2$$

下面来计算卷积层第 1 个节点的值，图 5.12 左侧是输入层，为 5×5 的图像，如图上浅色中间部分所示，而其外围为 0，因为我们规定边缘 0 填充值为 1，所以加了一圈 0。图中间一列是卷积层与输入连接权值，因为我们规定接收域为 3×3（超参数 $F=3$ ），而每个卷积层神经元分别与输入层 RGB 三个分量上对应（行、列值相同）的子区域相连接，在本例中，一个卷积层神经元与输入层神经元的连接数为 $3 \times 3 \times 3$ ，图中间部分为这些连接权值的值。卷积层神经元与输入层神经元连接的接收域为以其行、列为中心的 3×3 子区域，如图中加粗的部分。

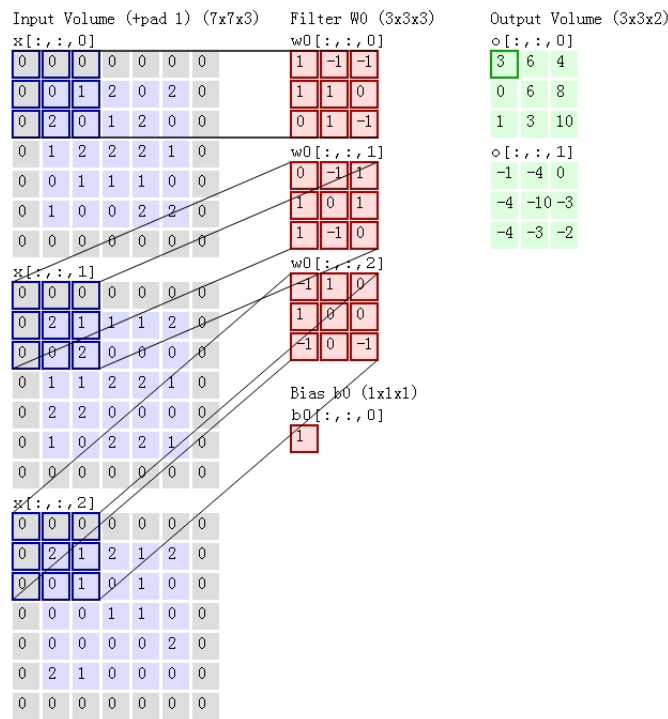


图 5.12 卷积操作示例 1

下面是求卷积操作，就是先将连接权值与对应输入层神经元输出值相乘，再求这些值之和，即卷积层神经元的输出值。计算公式如下：

$$\begin{aligned} o[0,0,0] &= (0 \times 1 + 0 \times (-1) + 0 \times (-1) + 0 \times 1 + 0 \times 1 + 1 \times 0 + 0 \times 0 + 2 \times 1 + 0 \times (-1)) + \\ &\quad (0 \times 0 + 0 \times (-1) + 0 \times 1 + 0 \times 1 + 2 \times 0 + 1 \times 1 + 0 \times 1 + 0 \times (-1) + 2 \times 0) + \\ &\quad (0 \times (-1) + 0 \times 1 + 0 \times 0 + 0 \times 1 + 2 \times 0 + 1 \times 0 + 0 \times (-1) + 0 \times 0 + 1 \times (-1)) + \\ &\quad 1(\text{bias}) \\ &= 3 \end{aligned}$$

所以卷积层神经元在深度为 0 切片(0,0)位置神经元输出值为 3。
因为我们规定步长为 2，所以卷积层深度为 0 切片(0,1)位置神经元，对应于输入层接收域就变为以(2,0)为中心的 3×3 子区域，对应区域为图 5.13 中粗线所示神经元。

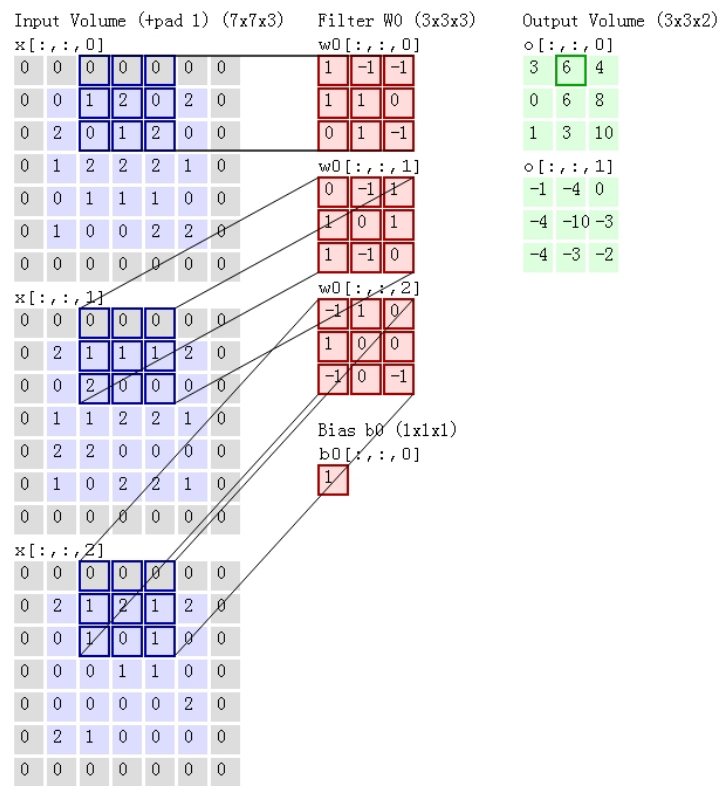


图 5.13 卷积操作示例 2

由权值共享知识可知，卷积层深度第 0 层切片上的神经元将共享其与输入层接收域的连接权值，因此中间的连接权值与上面的相同，可以用相同的方法求出卷积层深度第 0 层切片(0,1)位置神经元输出值。

图 5.14 显示的是卷积层深度第 0 层切片(0,2)位置，即边缘位置卷积操作的情形，注意此时的连接权值不变，对应的输入层接收域用粗线表示。

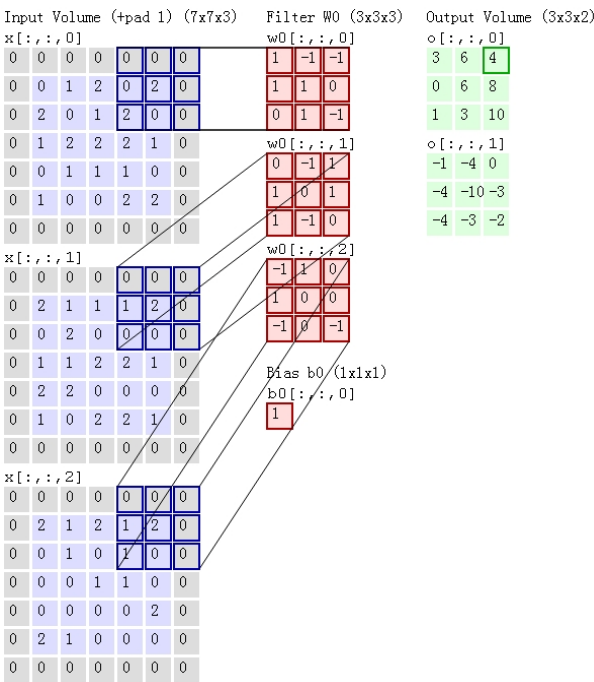


图 5.14 卷积操作示例 3

图 5.15 是卷积层深度第 0 层(2,2)位置神经元卷积计算过程，这个是卷积层深度第 0 层最后一个神经元。注意，由于权值共享，此时权值仍然没有改变，输入层中对应卷积操作的接收域神经元用粗线标出。

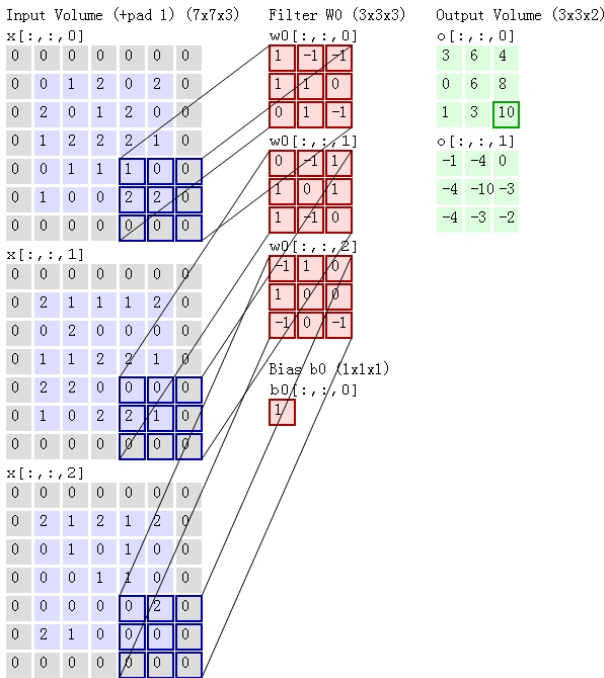


图 5.15 卷积操作示例 4

到目前为止，我们完成了卷积层深度第 0 层切片上所有神经元的输出值计算，下面我们来进行卷积层深度第 1 层神经元的输出值计算。

由于计算卷积层深度第 1 层神经元，因此其连接权值将发生变化，但是输入层对应的接收域却与深度第 0 层相应神经元的相同。如图 5.16 所示，计算卷积层深度第 1 层(0,0)位置神经元输出值。

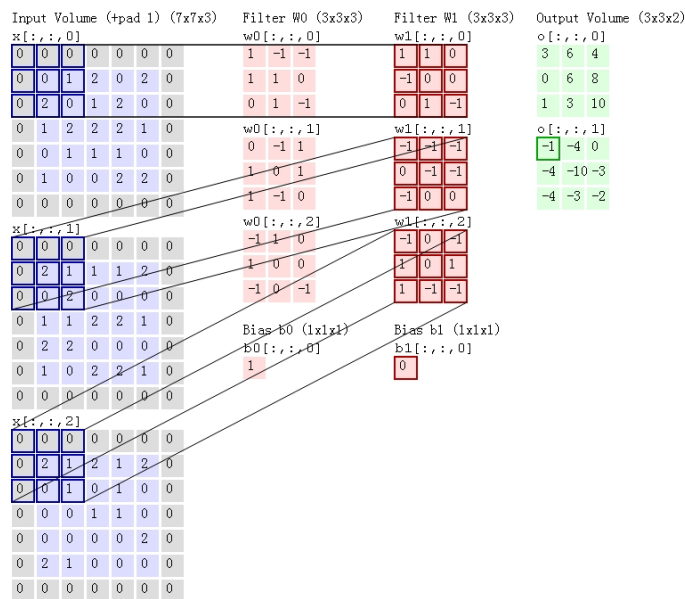


图 5.16 卷积操作示例 5

由图 5.16 可以看出，我们连接权值用的是 W1 而不是 W0。除所有连接权值和所求卷积层深度切片不同外，其余操作均与卷积层第 0 层相同。图 5.17 是第 1 层(2,2)位置神经元输出值计算示意图，这是卷积层最后一个神经元。

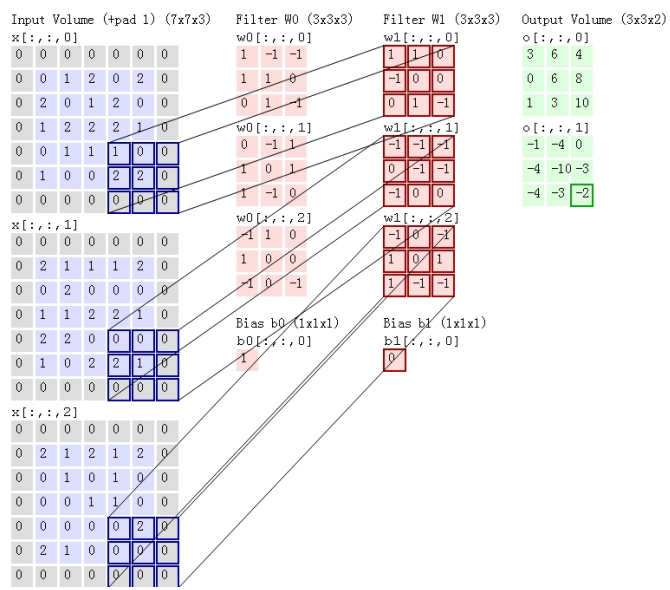


图 5.17 卷积操作示例 6

卷积层经过卷积操作后，会输出到 ReLU 层，其会对信号进行一个非线性变换，并输出到池化层。因为 ReLU 层既没有连接权值，也没有偏移量，所以不需要学习算法进行调整，比较简单，我们就不再介绍了。

从上面的分析我们不难看出，卷积操作是运算量很巨大的操作。卷积操作是卷积层神经元与输入层对应接收域神经元连接权值与对应输入层神经元输出值取点积，根据这个特点，我们可以将卷积操作转化为一个大的矩阵乘法，因为很多科学计算库都对矩阵乘法做了特别优化，因此可以大幅提高计算效率。

以卷积操作的例子为例，输入层为 $5 \times 5 \times 3$ ，即 $W1=5$ 、 $H1=5$ 、 $D1=3$ ，接收域为 3×3 ，所以卷积层超参数为：深度（滤波器个数） $K=D2=2$ ，步长 $S=2$ ，边缘 0 填充数量 $P=2$ ，因此卷积层尺寸为 $3 \times 3 \times 2$ 。

为了高效计算卷积操作，我们需要将卷积操作转换为矩阵操作。我们将输入层接收域中的神经元输出值展开拉直后，作为矩阵的一个列，这个操作叫 `im2col`。具体方法如下：卷积层神经元在输入层上的接收域为 $3 \times 3 \times 3$ ，表示在输入层每个深度切片上接收域为 3×3 子区域，共有 3 个深度层切片，这样就变成共有 27 个元素的列向量。卷积层共有 $3 \times 3 \times 2$ 个神经元，每个卷积层神经元均有 $3 \times 3 \times 3$ 个输入层接收域，因为步长 $S=2$ ，边缘 0 填充 $P=1$ ，宽度 $W1=5$ ，所以对卷积层深度方向上每个切片，由 $\frac{W1-F+2P}{S} + 1 = \frac{5-3+2 \times 1}{2} + 1 = 3$ 可知，共产生 3 个这样的列向量，由于输入图像宽度和高度相同，因此高度方面也可以产生 3 个，所以一共可以产生 $3 \times 3 = 9$ 个上述列向量，每个列向量维度为 $3 \times 3 \times 3 = 27$ ，将这些列向量组合在一起就形成了 27×9 的矩阵，将其记为 W_col 。

对于卷积层神经元与输入层接收域内神经元的连接权值，我们同样将其展开拉直，作为行向量 W_row 的一行，我们知道对每个卷积层神经元，这样的连接权值为 $3 \times 3 \times 3 = 27$ ，而由于权值共享，卷积层每个深度平面共享连接权值，因此一共有两个这样的权值组成的矩阵行，可以将其组合为 2×27 的矩阵。

这时卷积操作就变成了 $W_row \times W_col$ ，形成 $R^{2 \times 27} \times R^{27 \times 9} = R^{2 \times 9}$ 的矩阵，每行对应卷积层对应深度切片上所有神经元的输出值。

最后将 2×9 矩阵转化为按深度存储的卷积层神经元排列方式，即将 2×9 转换为 $3 \times 3 \times 2$ 。

上面的算法可以极大提高计算效率，但是却会占用更多的内存空间，因为输入层的接收域根据步长的不同会有很大程度的重叠，因此我们定义的列矩阵 W_col 会有很多重复元素占用额外的空间。但是由于这种算法可以利用科学计算库（如 Blas）等，而且还可以与池化层操作统一，因此在实际中被广泛采用。

为了提高卷积神经网络中的卷积计算效率，研究人员提出了多种模型，我们在这里仅向大家介绍两种典型的模型。

第一种是网中网（NIN）模型。在前面的讨论中，接收域的大小一般取 3×3 、 5×5 、 7×7 等，但是近年来有研究人员采用 1×1 的接收域。注意，由于输入层的深度为 3，所以即使是 1×1 的接收域，也是对颜色值 RGB 分量进行点积操作，也是进行了卷积操作。这种技术可以更好地识别接收域内的特征，也可以改善与其配合的全连接的宏网络的过拟合问题。

第二种是扩张卷积模型。在通常情况下，我们在进行卷积操作时是选择连续的输入层接收域内的神经元，这样如果我们的接收域为 3×3 ，则第二个卷积层在原始输入层的接收域就会扩大为 5×5 ，越往高层走，卷积层神经元在原始输入对应的接收域就越大。扩张卷积网络在进行卷积操作时，可以越过一些输入层接收域内的神经元，因此可以有效扩大卷积层在原始输入层上的接收域，减少卷积神经网络层次。

2. 池化层

在卷积神经网络中，我们通常周期性地在连续的卷积层之间插入一个池化层，这样做的目的是持续减少描述原始图像的特征，从而减少网络所需参数、网络训练所需运算量，也可以避免过拟合。

在实际应用中，采用最多的是最大值池化方法。对经过 ReLU 层或直接卷积的结果，对深度方面每个切片单独做最大池化操作。最常见的是采用 2×2 最大池化滤波器，取输入 2×2 区域中的最大值，作为池化层神经元的输出，步长为 2，移动下一个 2×2 区域，继续做最大池化操作。由此可见，每经过一次最大池化操作，将舍弃原图像中 75% 的信息，实现了信息的压缩。而我们前文所述的大小、旋转不变化等特性，在很大程度上也是由于我们的最大池化操作所产生的。最大池化的规则如下。

接收来自 ReLU 层或卷积层的输入信号，维度为： $W1 \times H1 \times D1$ 。

池化层超参数：以最大池化为例。

池化滤波器大小 F ：例如 $F=2$ ，在 2×2 区域内取最大值。

池化步长 S ：例如 $F=2$ 时，如 $S=2$ 则连续取池化，但是不重合。

池化层神经元组织形式： $W2 \times H2 \times D2$ 。

$$W2 = \frac{W1 - F}{S} + 1$$

$$H2 = \frac{H1 - F}{S} + 1$$

$$D2 = D1$$

进行池化时没有其他操作，因此没有参数，不需要学习算法对其进行调整，并且通常也不需要边缘进行 0 填充。

在上面的介绍中，我们是以最大池化为例进行介绍的，在实际应用中，特别是前几年，使用比较多的方法还有平均池化，就是取池化区域中像素点的平均值，但是后来发现，采用平均池化的性能不好，目前已经较少采用了。

下面以一个具体实例来向大家演示最大池化的操作过程。假设从 ReLU 层发出的信号为 $4 \times 4 \times 2$ ，即切片分辨率为 4×4 ，深度为两层，如图 5.18 中左侧所示，我们设最大池化滤波器大小为 2×2 ，步长为 2，图中左侧用不同颜色标示出最大池化滤波器在输入信号上的作用区域，图中右侧用不同颜色标示出不同最大池化滤波器输出结果。

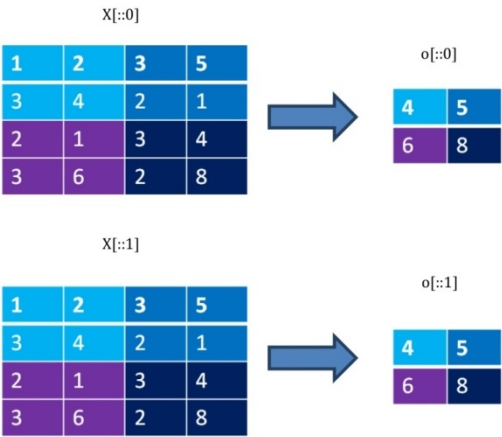


图 5.18 最大池化示例

如上图所示，每个输入层切片为 4×4 ，共有 4 个不同的最大池化滤波器作用域，用左上角四个单元、左下角四个单元、右上角四个单元、右下角四个单元表示，池化步长为 2，因此池化区域不重叠，对应池化结果如图中右侧所示，分别用不同颜色一一对应。

3. 全连接层

卷积神经网络中的全连接层的组织结构与多层感知器模型相同，同时，如果针对 MNIST 手写数字识别任务，其输出层同样为 softmax 层神经元组成。

在实际应用中，经常会将全连接层转化为卷积层，或者将卷积层转换为全连接层。我们将分别来讨论。

首先来看将卷积层转换为全连接层，那么连接权值矩阵将是一个稀疏矩阵，只有少数位置值不为零，因为卷积层具有层间稀疏连接的特性，只有对应接收域的神经元的连接块权值才不为零，在这些接收域对应的连接块中，很多块的权值是彼此相等的，因为在卷积层深度方向同一切片具有权值共享特性。当将卷积层转换为全连接层后，卷积神经网络就变成普通的多层感知器模型。有研究人员采用卷积神经网络先对网络进行训练，训练结束后再将卷积层变为全连接层，转化为普通多层感知器模型，然后再对网络进行微调，这在实践中取得了较好的效果。

同样，我们也可以将全连接层转换为卷积层。我们以 $224 \times 224 \times 3$ 的彩色图像为例，如果卷积层深度（滤波器数量）为 $K=512$ ，最大池化层采用 2×2 且步长为 2，经过五次卷积→ReLU→最大池化之后，最终的输出为 $7 \times 7 \times 512$ ，即共 25088 个神经元节点，然后连接到具有 4096 个神经元（ReLU）的全连接层，再连接到具有 4096 个神经元（ReLU）节点的全连接层，最后连接到具有 1000 个 softmax 激活函数的输出层神经元，计算对应 ImageNet 图片 1000 个类别的出现概率。

最后池化层输出为 $7 \times 7 \times 512 = 25088$ ，卷积层大小为： $\frac{W1-F+2P}{s} + 1 = \frac{7-7+2 \times 0}{1} + 1 = 1$ ，所以采用 7×7 卷积核，边缘 0 填充 $P=0$ ，深度为 4096，因此转换后的卷积层为 $1 \times 1 \times 4096$ 。

第二个全连接层采用 1×1 卷积核，深度同样为 4096，因此形成的卷积层内核为 $1 \times 1 \times 4096$ 。

对于输出层，采用 1×1 卷积核，深度设定为 1000，因此形成的卷积层为 $1 \times 1 \times 1000$ 。

以最后的最大池化层到第一个全连接层为例，如果采用全连接层，需要 102764544 ($7 \times 7 \times 512 \times 4096 + 4096$) 个参数，而转换为卷积层之后，由权值共享可以得到每个深度切片共享与前一层 7×7 接收域的连接权值，因此总参数为 204800 ($7 \times 7 \times 4096 + 4096$) 个，减少了 500 多倍，可以极大地提高训练效率。

将全连接层转化为卷积层不仅可以减少参数数量，还可以更加高效地处理不同分辨率的图像。接上例，如果此时处理的图像是 384×384 ，卷积层深度（滤波器数量）为 $K=512$ ，最大池化层采用 2×2 且步长为 2，经过五次卷积→ReLU→最大池化之后，最终的输出为：

$12 \times 12 \times 512$ ，下一个卷积层大小为： $\frac{W1-F+2P}{s} + 1 = \frac{12-7+2 \times 0}{1} + 1 = 6$ ，则第一个全连接

层的深度为 4096，卷积核为 6，所以形成的卷积层为 $6 \times 6 \times 4096$ 。第二个全连接层也是 $6 \times 6 \times 4096$ ，输出层就变为 $6 \times 6 \times 1000$ 。

上述讨论说明，我们通常会将图像原始尺寸放大，从而利用不同的空间关系，这样就可以同时计算一个类别多个出现概率的值，最后以其平均值作为结果。目前在 MNIST 手写数字识别字符集上，最好的结果在测试样本集上的误差 0.21%，就是利用这种技术得到的。

在实际应用中，我们可以采用网络手术（Net Surgery）技术对网络架构进行修改，从而使其符合需求。

5.1.3 卷积神经网络设计

1. 网络架构设计

通常卷积神经网络的架构如图 5.19 所示。

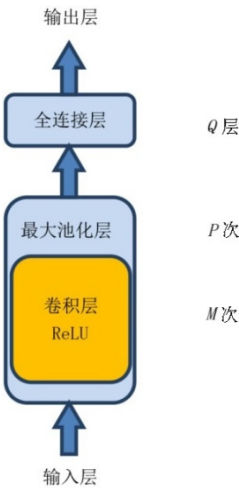


图 5.19 卷积神经网络架构

图像先进入输入层，具体流程如下。

- (1) 卷积操作：进入卷积层进行卷积，再进入 ReLU 层进行非线性变换。
- (2) 重复 M 次卷积操作。
- (3) 池化操作：进入最大池化层，进行向下采样。以上步骤统称为卷积池化操作。
- (4) 进行 P 次卷积池化操作。
- (5) 进入全连接层，全连接层可以有 Q 层。
- (6) 进入输出层。

下面给出一个具体的网络的例子，给大家一个感性的认识。

如图 5.20 所示，信号经过输入层先到达卷积层 1，到 ReLU 层 1，再经卷积层 2 和 ReLU 层 2 到达卷积层 3 和 ReLU 层 3，再进入最大池化层 1。然后进入卷积层 4 和 ReLU 层 4，再进入卷积层 5 和 ReLU 层 5，接着到卷积层 6 和 ReLU 层 6，再进入最大池化层 2。最后进入全连接层 1 和全连接层 2，再进入输出层产生输出结果。在这个具体的网络架构中，我们可以得到 $M=6$ 、 $P=2$ 、 $Q=2$ 。

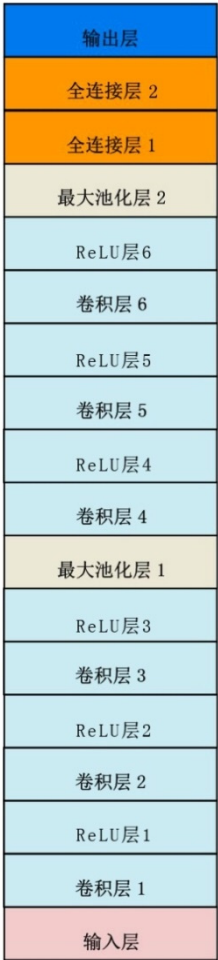


图 5.20 卷积神经网络架构示例

与多层感知器模型类似，在设计卷积神经网络时，同样需要选择是建立浅而宽的网络呢？还是建立窄而深的网络呢？例如，我们将三个以 3×3 为接收域的卷积层加 ReLU 层级联起来，那么在卷积层 1 中，对应输入层的接收域为 3×3 区域，但是在卷积层 2 中，由于其直接接收域为卷积层 1 上的 3×3 区域，所以在输入层的原始接收域为 5×5 区域。同理，在卷积层 3，在输入层的原始接收域就为 7×7 区域。由此可见，卷积层叠加越多，最上面的卷积层在输入层的原始接收域就会越大。同样，我们也可以用了一个 7×7 为接收域的卷积层，其在输入层上的接收域与前述三个叠加的卷积层相同。

实验证明，用叠加的卷积层加小接收域的效果要优于单层大接收域卷积层。首先，三个卷积层叠加，在每次卷积操作之后，会做一个非线性变换，而只有一层卷积层，则只做了卷积操作。因为我们知道卷积操作就是点积操作，是一个线性变换，因此在特征表现能力上来看，三个叠加卷积层的特征表现能力要优于单个卷积层。其次，假设卷积层深度（滤波器数量）均为 10，采用单个卷积层所需参数为： $7 \times 7 \times 10 = 490$ ，而三个卷积层叠加则则为： $3 \times 3 \times 10 + 3 \times 3 \times 10 + 3 \times 3 \times 10 = 270$ 。由此可见，在三个卷积层叠加的情况下，网络所需参数也将少于单个卷积层。综上所述，采用多个卷积层级联方式，可以增加模型特征的表现能力，需要的学习参数更少，因此实践中通常用小的卷积层级联。当然这种方法也有缺点，就是在执行 BP 算法时，需要更大的内存。

2. 惯例

输入层主要是接收原始图像像素值数据，彩色图像深度通常为 3，黑白图像深度为 1，视频或医学 fMRI 影像的深度可能为四维或以上，但是原理是相同的。对于图像分辨率，为了简单起见，均取 2 的整数倍，同时宽度和高度取相同数值，典型的值有：28（MNIST 手写数字识别）、32（CIFAR-10）、64、96、224（ImageNet）、384、512 等。

卷积层一般用小的接收域，例如 3×3 、 5×5 。卷积操作步长 $S=1$ 。为了保持原来的尺寸，边缘需要 0 填充，一般接收域为 3×3 时，边缘 0 填充 $P=1$ ；接收域为 5×5 时，边缘 0 填充 $P=2$ 。一般来讲，除了直接与输入层相连的卷积层，一般不使用大的接收域。

池化层一般采用最大池化操作，取 2×2 滤波器尺寸，取其中的最大值作为池化层神经元输出值。

5.1.4 迁移学习和网络微调

在实际应用中，我们很少从头开始创建并训练一个卷积神经网络。一是因为训练一个用于 ImageNet 图像识别的卷积神经网络，即使是在装有 GPU 的云平台上，也需要两三周的时间。二是在实际中也很难找到如此巨大的图像样本库用于训练卷积神经网络。更常见的做法是，选择一个训练好的卷积神经网络进行学习迁移，例如已经训练好的 ImageNet 数据集上的卷积神经网络。

采用预训练的卷积神经网络，我们有两种策略。

第一种策略是将预训练好的卷积神经网络去掉最后一层输出层，将其输出作为图像的特征提取器，然后用这些特征作为输入重新训练自己的分类器，从而完成任务。以 AlexNet 为例，其倒数第二层全连接层有 4096 个神经元，我们可以重新构造一个神经网络，其输入信号的维度为 4096，再将预训练好的卷积神经网络直接接入到新网络的输入层，然后用自己的数据重新进行训练。

第二种策略是进行网络微调。网络微调是指，我们不仅重新训练预训练好的卷积神经网络的模式分类层，也重新训练前面几层。根据研究发现，卷积神经网络最底层的卷积层、ReLU 层和最大池化层主要完成初级图像处理工作，例如边缘检测、区域生长等，而越往上，高层的卷积层、ReLU 层和最大池化层更多地会与特定数据集的特征相关，因此有微调的必要性。所以在实际应用中，我们会用自己的数据集来训练预训练好的卷积神经网络，并规定最底下几层卷积层、ReLU 层和最大池化层的参数不变，只有顶部的几层参与学习和参数调整，最后得到完整的网络。

那么在实际问题中，我们到底该采用哪种策略呢？在这里我们主要需要考虑两种因素：数据集的大小和图像的性质，与预训练卷积神经网络训练样本集是否相似，例如 ImageNet 主要是自然景物，如果要识别的图像是 X 光片，那么数据集与 MNIST 手写数字识别数据集就不相像了。下面我们将分不同的情况进行讨论。

- ❑ 训练样本集较小，而且与预训练的卷积神经网络的训练样本集相似。这种情况下应该考虑将预训练的卷积神经网络作为图像的特征提取器，将预训练的卷积神经网络原输出层的输入作为新的模式分类器的输入，从头开始训练模式分类器。因为在训练样本集较小的情况下，如果采用网络微调方式，在追求好的分类结果时很容易造成过拟合现象，但是由于训练样本集与预训练的卷积神经网络原始训练样本集相似，我们有理由相信，预训练的卷积神经网络可以很好地提取出合适的图像特征，因此采用将预训练卷积神经网络作为特征提取器是适当的。
- ❑ 训练样本集很大，而且与预训练的卷积神经网络的原始训练样本集类似。这种情况下应该采用网络微调方式。因为训练样本足够多，就可以避免出现过拟合现象，这时可以固定底层的参数，只对顶层参数进行调整。
- ❑ 训练样本集很小，而且与原始训练样本集不相似。在这种情况下，靠近顶层的卷积层、全连接层等，会带有原始训练样本集的特征，所以最好不采用。因此需要选择某个靠近底层的最大池化层，将其作为输入，接入自己的模式分类器网络，用训练样本集重新训练网络。
- ❑ 训练样本集很大，而且与原始训练样本集不相似。在这种情况下，可以训练一个全新的网络。但是以一个预训练好的卷积神经网络作为起点，也可以提高训练效果。

还需要注意一点，如果使用预训练的卷积神经网络，学习率要取得适当小一些，否则可能造成不收敛。

下面将讨论怎样采用 Theano 框架实现卷积神经网络，并将其用于 MNIST 手写数字识别任务。

5.2 卷积神经网络的 TensorFlow 实现

我们先来看一个简单的例子，看看对一幅图像进行一次卷积操作后会产生怎样的效果，使大家对卷积操作有一个感性的认识。

在具体讲解代码之前，先看一下 TensorFlow 中与卷积和池化相关的函数，首先看卷积函数。

```
1 tf.nn.conv2d(input, filter, strides, padding, use_cudnn_on_gpu=None, data_
format=None, name=None)
```

- ❑ **input**: 卷积操作的输入信号，形状为：[迷你批次内样本序列号，宽度，高度，信道数量]。
- ❑ **filter**: 滤波器形状，形状为：[滤波器宽度，滤波器高度，输入信号通道数，输出信号通道数（特征图数量）]。
- ❑ **strides**: 各个方向的卷积步长，形状为：[1，宽度方向步长，高度方向步长，1]，通常取宽度和高度方向上的步长一致。
- ❑ **padding**: 指定边缘 0 填充形式，这里有两个值，一是“SAME”：采用边缘 0 填充，例如 3×3 滤波器在边缘填充一个像素，而 5×5 滤波器在边缘填充两个像素，这样做完卷积操作后，图像的尺寸不变；二是“VALID”：边缘不采用 0 填充，将忽略边缘像素值，卷积操作将使图像尺寸减小。

其他参数和我们目前的关系不大，就不在此做介绍了。

在下面的代码中，我们将利用随机数初始化一个卷积层，对 256×256 图像进行卷积操作，分别叠加 ReLU 神经元和 Sigmoid 神经元，并将两种情况得出的结果特征图进行比较，代码如下（完整的代码参见 <https://github.com/dlp.git> 的 book/chp05/conv_demo.py）：

```
1 def startup(self):
2     print('Convolutional Demo')
3     img_file = 'datasets/wolfs.jpg'
4     img_raw = io.imread(img_file)
5     img = img_raw.astype(np.float32) / 255.0
6     x = img.reshape([1, 256, 256, 3])
7     X = tf.placeholder(shape=[None, 256, 256, 3], dtype=tf.float32)
8     W_1 = tf.Variable(tf.truncated_normal(shape=[3, 3, 3, 2], mean=0.0, stddev=0.1))
9     b_2 = tf.Variable(tf.zeros([256, 256, 2]))
10    z_2 = tf.nn.conv2d(X, W_1, strides=[1, 1, 1, 1], padding='SAME') + b_2
11    a_2_relu = tf.nn.relu(z_2)
12    a_2_sigmoid = tf.nn.sigmoid(z_2)
13    with tf.Session() as sess:
14        sess.run(tf.global_variables_initializer())
15        rst_relu = sess.run(a_2_relu, feed_dict={X: x})
16        rst_sigmoid = sess.run(a_2_sigmoid, feed_dict={X: x})
17        fm1 = rst_relu[0, :, :, 0]
18        fm2 = rst_relu[0, :, :, 1]
19        fm3 = rst_sigmoid[0, :, :, 0]
20        fm4 = rst_sigmoid[0, :, :, 1]
21        plt.figure(1)
22        plt.subplot(231)
23        plt.imshow(img)
24        plt.axis('off')
25        plt.title('origin')
```



```

26     plt.subplot(232)
27     plt.imshow(fm1)
28     plt.axis('off')
29     plt.title('ReLU fm1')
30     plt.subplot(233)
31     plt.imshow(fm2)
32     plt.axis('off')
33     plt.title('ReLU fm2')
34     plt.subplot(235)
35     plt.imshow(fm3)
36     plt.axis('off')
37     plt.title('sigmoid fm1')
38     plt.subplot(236)
39     plt.imshow(fm4)
40     plt.axis('off')
41     plt.title('sigmoid fm2')
42     plt.show()

```

第 3 行：指定图像为网上著名的 3 只狼和月亮的 256×256 图像。

第 4 行：读出图像内容，这时 `img_raw` 为[256, 256, 3]的数组。

第 5 行：将数组元素类型转换为 `float32`，然后除以 255.0，使其成为 0~1 的浮点数（原始图像中像素值为 0~255 的整数）。

第 6 行：生成一个可以用于卷积操作的样本，形状为[1, 256, 256, 3]。

第 7 行：定义表示输入信号迷你批次的设计矩阵 `X`，第一维为 `None`，在运行时根据迷你批次大小赋值。

第 8 行：定义输入层到卷积层的连接权值 `W_1`，滤波器尺寸为 3×3，输入信号的维度为 3（具有 RGB 通道的彩色图像），输出信号通道为 2，即卷积操作后会生成两个 256×256 的特征图。

第 9 行：生成卷积层神经元偏置值。

第 10 行：定义卷积操作，首先进行宽高方向步长均为 1，边缘采用 `SAME` 填充，以 `W_1` 为滤波器的卷积操作，然后加上对应的偏置值。

第 11 行：采用 `ReLU` 神经元时卷积层的输出信号。

第 12 行：采用 `Sigmoid` 神经元时卷积层的输出信号。

第 13 行：启动 `TensorFlow` 会话。

第 14 行：初始化所有变量。

第 15 行：采用 `ReLU` 激活函数，计算卷积层输出值。

第 16 行：采用 `Sigmoid` 激活函数，计算卷积层输出值。

第 17 行：取出采用 `ReLU` 激活函数时，卷积层得到的第 1 张特征图（共 2 张）。

第 18 行：取出采用 `ReLU` 激活函数时，卷积层得到的第 2 张特征图（共 2 张）。

第 19 行：取出采用 `Sigmoid` 激活函数时，卷积层得到的第 1 张特征图（共 2 张）。

第 20 行：取出采用 `Sigmoid` 激活函数时，卷积层得到的第 2 张特征图（共 2 张）。

第 21 行：初始化 `matplotlib` 绘图库。

第 22~25 行：在 2 行 3 列网格的第 1 行第 1 列绘制原始图片。

第 26~29 行：在 2 行 3 列网格的第 1 行第 2 列绘制使用 `ReLU` 激活函数时的第 1 张特征图。

第 30~33 行：在 2 行 3 列网格的第 1 行第 3 列绘制使用 ReLU 激活函数时的第 2 张特征图。

第 34~37 行：在 2 行 3 列网格的第 2 行第 2 列绘制使用 Sigmoid 激活函数时的第 1 张特征图。

第 38~41 行：在 2 行 3 列网格的第 2 行第 3 列绘制使用 Sigmoid 激活函数时的第 2 张特征图。

第 42 行：绘制所有图像。

运行上面的程序，结果如图 5.21 所示。

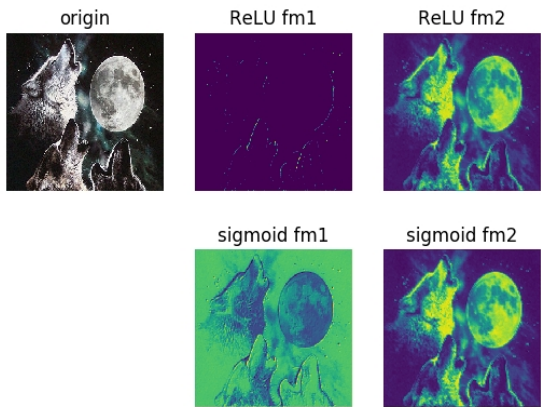


图 5.21 卷积操作效果示例

如图 5.21 所示，在未经过训练的滤波器的情况下，卷积层的输出近似于取出图像的边缘。事实上，在训练好的网络中，卷积层的输出也是边缘提取的效果。这与理论研究得出的结果一致，人类的初级视觉主要也是边缘提取和区域分割，这证明卷积神经网络在生物学上还是有其合理性的。

在对卷积操作有了感性认识之后，我们开始正式用 TensorFlow 来实现卷积神经网络，并将其用于 MNIST 手写数字识别任务。先带领大家看一下卷积神经网络之父 Yann LeCun 在 1998 年的原始论文，了解一下 LeNet5 的网络结构，同时也看一下，我们怎样通过论文来跟踪国际上最新的研究成果，并通过复现这些研究成果，深入理解其中的原理，尽快应用到实际项目中去。

5.2.1 模型搭建

对卷积神经网络的研究是由 Yann LeCun 在 1998 年发现的一篇论文引起的 (http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf)。现在网络上关于 LeNet5 的内容的源头均出自这篇论文，读者如果英文水平还可以的话，可以尝试阅读一下这篇经典论文。

在这篇论文中，Yann LeCun 提出了如图 5.22 所示的卷积神经网络结构。

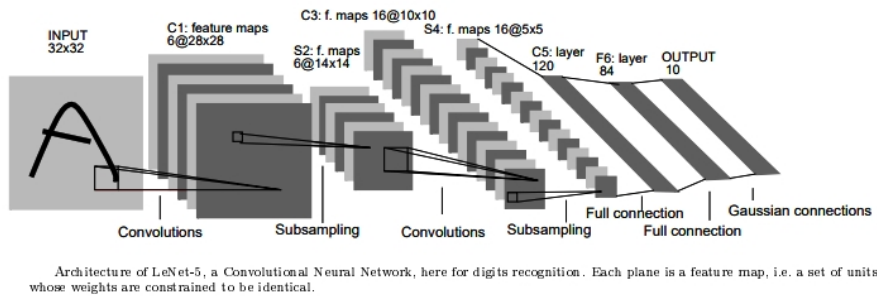


图 5.22 LeNet5 网络架构

关于网络架构的描述，原始论文如图 5.23 所示。

This section describes in more detail the architecture of LeNet-5, the Convolutional Neural Network used in the experiments. LeNet-5 comprises 7 layers, not counting the input, all of which contain trainable parameters (weights). The input is a 32x32 pixel image. This is significantly larger than the largest character in the database (at most 20x20 pixels centered in a 28x28 field). The reason is that it is desirable that potential distinctive features such as stroke end-points or corner can appear in the center of the receptive field of the highest-level feature detectors. In LeNet-5 the set of centers of the receptive fields of the last convolutional layer (C3, see below) form a 20x20 area in the center of the 32x32 input. The values of the input pixels are normalized so that the background level (white) corresponds to a value of -0.1 and the foreground (black) corresponds to 1.175. This makes the mean input roughly 0, and the variance roughly 1 which accelerates learning [46].

In the following, convolutional layers are labeled Cx, subsampling layers are labeled Sx, and fully-connected layers are labeled Fx, where x is the layer index.

Layer C1 is a convolutional layer with 6 feature maps. Each unit in each feature map is connected to a 5x5 neighborhood in the input. The size of the feature maps is 28x28 which prevents connection from the input from falling off the boundary. C1 contains 156 trainable parameters, and 122,304 connections.

Layer S2 is a sub-sampling layer with 6 feature maps of size 14x14. Each unit in each feature map is connected to a 2x2 neighborhood in the corresponding feature map in C1. The four inputs to a unit in S2 are added, then multiplied by a trainable coefficient, and added to a trainable bias. The result is passed through a sigmoidal function. The 2x2 receptive fields are non-overlapping, therefore feature maps in S2 have half the number of rows and column as feature maps in C1. Layer S2 has 12 trainable parameters and 5,880 connections.

Layer C3 is a convolutional layer with 16 feature maps. Each unit in each feature map is connected to several 5x5 neighborhoods at identical locations in a subset of S2's feature maps. Table I shows the set of S2 feature maps

combined by each C3 feature map. Why not connect every S2 feature map to every C3 feature map? The reason is twofold. First, a non-complete connection scheme keeps the number of connections within reasonable bounds. More importantly, it forces a break of symmetry in the network. Different feature maps are forced to extract different (hopefully complementary) features because they get different sets of inputs. The rationale behind the connection scheme in table I is the following. The first six C3 feature maps take inputs from every contiguous subsets of three feature maps in S2. The next six take input from every contiguous subset of four. The next three take input from some discontinuous subsets of four. Finally the last one takes input from all S2 feature maps. Layer C3 has 1,516 trainable parameters and 151,600 connections.

Layer S4 is a sub-sampling layer with 16 feature maps of size 5x5. Each unit in each feature map is connected to a 2x2 neighborhood in the corresponding feature map in C3, in a similar way as C1 and S2. Layer S4 has 32 trainable parameters and 2,000 connections.

Layer C5 is a convolutional layer with 120 feature maps. Each unit is connected to a 5x5 neighborhood on all 16 of S4's feature maps. Here, because the size of S4 is also 5x5, the size of C5's feature maps is 1x1: this amounts to a full connection between S4 and C5. C5 is labeled as a convolutional layer, instead of a fully-connected layer, because if LeNet-5 input were made bigger with everything else kept constant, the feature map dimension would be larger than 1x1. This process of dynamically increasing the size of a convolutional network is described in the section Section VII. Layer C5 has 48,120 trainable connections.

Layer F6, contains 84 units (the reason for this number comes from the design of the output layer, explained below) and is fully connected to C5. It has 10,164 trainable parameters.

As in classical neural networks, units in layers up to F6 compute a dot product between their input vector and their weight vector, to which a bias is added. This weighted sum, denoted a_i for unit i , is then passed through a sigmoid squashing function to produce the state of unit i , denoted by x_i :

$$x_i = f(a_i) \tag{5}$$

The squashing function is a scaled hyperbolic tangent:

$$f(a) = A \tanh(Sa) \tag{6}$$

where A is the amplitude of the function and S determines its slope at the origin. The function f is odd, with horizontal asymptotes at $+A$ and $-A$. The constant A is chosen to be 1.7159. The rationale for this choice of a squashing function is given in Appendix A.

Finally, the output layer is composed of Euclidean Radial Basis Function units (RBF), one for each class, with 84 inputs each. The outputs of each RBF unit y_i is computed as follows:

$$y_i = \sum_j (x_j - w_{ij})^2. \tag{7}$$

图 5.23 Yann LeCun 的原始论文

我们可以看出，原始论文中对网络结构的描述还是很清晰的，我们完全可以按照论文的描述搭建出这个网络。下面我们就从头开始搭建这个网络。

在这一部分，我们将按照论文的相关章节搭建一个 LeNet5 网络，我们会对原始架构进行一些修改，因为我们现在对算法有了更深入的了解，具体的更改将在代码中进行讲解，代码如下：

```
def build_model(self):
1   print('build convolutional neural network')
2   X = tf.placeholder(shape=[None, 28, 28, 1], dtype=tf.float32)
3   self.model['X'] = X
4   y = tf.placeholder(shape=[None, self.k], dtype=tf.float32)
5   self.model['y'] = y
6   keep_prob = tf.placeholder(tf.float32) #Dropout 失活率
7   self.model['keep_prob'] = keep_prob
8   # 第2层第1个卷积层 c1
9   W_1 = tf.Variable(tf.truncated_normal(shape=[5, 5, 1, 6], mean=0.0,
stddev=0.1))
10  self.model['W_1'] = W_1
11  b_2 = tf.Variable(tf.zeros([28, 28, 6]))
12  self.model['b_2'] = b_2
13  z_2 = tf.nn.conv2d(X, W_1, strides=[1, 1, 1, 1], padding='SAME') + b_2
14  self.model['z_2'] = z_2
15  a_2 = tf.nn.relu(z_2)
16  self.model['a_2'] = a_2
17  # 第3层第1个最大池化层
18  m_3 = tf.nn.max_pool(a_2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
19  self.model['m_3'] = m_3
20  # 第4层第2个卷积层 C2
21  W_4 = tf.Variable(tf.truncated_normal(shape=[5, 5, 6, 16], mean=0.0, stddev=0.1))
22  self.model['W_4'] = W_4
23  b_4 = tf.Variable(tf.zeros([10, 10, 16]))
24  self.model['b_4'] = b_4
25  z_4 = tf.nn.conv2d(m_3, W_4, strides=[1, 1, 1, 1], padding='VALID') + b_4
26  self.model['z_4'] = z_4
27  a_4 = tf.nn.relu(z_4)
28  self.model['a_4'] = a_4
29  # 第5层第2个最大池化层
30  m_5 = tf.nn.max_pool(a_4, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
31  self.model['m_5'] = m_5
32  # 第6层第3个卷积层
33  W_5 = tf.Variable(tf.truncated_normal(shape=[5, 5, 16, 120], mean=0.0, stddev=0.1))
34  self.model['W_5'] = W_5
35  b_6 = tf.Variable(tf.zeros([1, 1, 120]))
36  self.model['b_6'] = b_6
37  z_6_raw = tf.nn.conv2d(m_5, W_5, strides=[1, 1, 1, 1], padding='VALID') + b_6
38  self.model['z_6_raw'] = z_6_raw
39  z_6 = tf.reshape(z_6_raw, [-1, 120])
40  self.model['z_6'] = z_6
41  a_6 = tf.nn.relu(z_6)
42  self.model['a_6'] = a_6
43  a_6_dropout = tf.nn.dropout(a_6, keep_prob)
44  self.model['a_6_dropout'] = a_6_dropout
45  # 第7层第1个全连接层
46  W_6 = tf.Variable(tf.truncated_normal(shape=[120, 84], mean=0.0, stddev=0.1))
47  self.model['W_6'] = W_6
48  b_7 = tf.Variable(tf.zeros([84]))
49  self.model['b_7'] = b_7
```

```

50 z_7 = tf.matmul(a_6_dropout, W_6) + b_7
51 self.model['z_7'] = z_7
52 a_7 = tf.nn.relu(z_7)
53 self.model['a_7'] = a_7
54 a_7_dropout = tf.nn.dropout(a_7, keep_prob)
55 self.model['a_7_dropout'] = a_7_dropout
56 # 第8层第2个全连接层
57 W_7 = tf.Variable(tf.truncated_normal(shape=[84, 10], mean=0.0, stddev=0.1))
58 self.model['W_7'] = W_7
59 b_8 = tf.Variable(tf.zeros([10]))
60 self.model['b_8'] = b_8
61 z_8 = tf.matmul(a_7_dropout, W_7) + b_8
62 self.model['z_8'] = z_8
63 y_ = tf.nn.softmax(z_8)
64 self.model['y_'] = y_
65 #训练部分
66 cross_entropy = tf.reduce_mean(-tf.reduce_sum(y * tf.log(y_),
67                                             reduction_indices=[1]))
68 self.model['cross_entropy'] = cross_entropy
69 #train_step = tf.train.AdagradOptimizer(0.3).minimize(cross_entropy)
70 loss = cross_entropy + self.lanmeda*(tf.reduce_sum(W_1**2) +
71                                     tf.reduce_sum(W_4**2) +
72                                     tf.reduce_sum(W_5**2) +
73                                     tf.reduce_sum(W_6**2) +
74                                     tf.reduce_sum(W_7**2))
75 self.model['loss'] = loss
76 train_step = tf.train.AdamOptimizer(learning_rate=0.001, beta1=0.9,
77                                     beta2=0.999, epsilon=1e-08, use_locking=False,
78                                     name='Adam').minimize(loss)
79 self.model['train_step'] = train_step
80 correct_prediction = tf.equal(tf.argmax(y_, 1), tf.argmax(y, 1))
81 self.model['correct_prediction'] = correct_prediction
82 accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
83 self.model['accuracy'] = accuracy
84 return X, y_, y, keep_prob, cross_entropy, train_step, \
85         correct_prediction, accuracy

```

第2行：定义输入信号的 placeholder 为四维张量[迷你批次样本序号，图像宽度，图像高度，通道数（黑白图像为 1，彩色图像为 3）]。

第3行：将其保存到 model 属性中。

第4行：定义输出标签正确结果的 placeholder，为二维张量[迷你批次样本序号，识别结果（表示 0~9 的 10 维向量）]。

第5行：将其保存到 model 属性中。

第6行：定义采用 Dropout 调整技术时，全连接层神经元保留比率 keep_prob 的 placeholder。

第7行：将 keep_prob 保存到 model 属性中。

第9行：定义第一个卷积层的滤波器组 W_1。这里需要重点说明一下，在论文的架构中，输入信号采用的是 32×32 的尺寸，也就是说需要在 28×28 原始图像边缘添加 2 个像素的边缘，卷积操作时采用 VALID 边缘填充模式（不进行 0 填充，因为已经填充到了原始图像中）。我们觉得这是没有必要的，可以使用原始图像 28×28 的分辨率作为输入。在卷积操

作时，采用 SAME 边缘填充模式，这样算法会自动在边缘进行 0 填充，效果与论文中的结果是一样的。滤波器的尺寸为 5×5，输入信号通道数为 1，输出通道数为 6，即会形成 6 个特征图。采用均值为 0.0、标准差为 0.1 的正态分布的随机数初始化滤波器组。注意，这里的滤波器组实际上就是多层感知器模型中的连接权值。

第 10 行：将 W_1 保存到 model 属性中。

第 11 行：定义第一卷积层神经元的偏置值 b_2，由于经过卷积操作后会得到 6 张 28×28 的特征图，所以偏置值 b_2 为[28, 28, 6]的张量。

第 12 行：将 b_2 保存到 model 属性中。

第 13 行：计算经过卷积操作后，卷积层神经元的输入信号为 z_2；卷积操作的卷积为上面定义的滤波器组 W_1，尺寸为 5×5，共 6 个；卷积操作宽度和高度方向步长均为 1；采用 SAME 边缘填充方式，即在边缘加 2 个像素 0 填充，这样可以保持图像的原始尺寸不变。根据公式得出：

$$w_{\text{conv}} = \frac{w_{\text{in}} - F + 2P}{S} + 1 = \frac{28 - 5 + 2 \times 2}{1} + 1 = 28$$

第 14 行：将 z_2 保存到 model 属性中。

第 15 行：采用 ReLU 神经元计算第一个卷积层的输出 a_2。

第 16 行：将 a_2 保存到 model 属性中。

第 18 行：定义第一个最大池化层，池化尺寸为 2×2；在宽度和高度方向的步长均为 2；边缘采用 SAME 0 填充方式，这里边缘 0 填充值为 2。根据公式得出：

$$W2 = \frac{W1 - F}{S} + 1 = \frac{28 - 2}{2} + 1 = 14$$

这样就可以得到论文中 6 张 14×14 的特征图了。

第 19 行：将 m_3 保存到 model 属性中。

第 21 行：定义第 2 个卷积层的卷积核，尺寸为 5×5，输入信号通道为 6，输出信号特征图数量为 16。从这里可以看到，在一般的卷积网络中，通常是特征图尺寸越来越小，但是通道数却越来越多。然后以均值为 0.0、标准差为 0.1 的正态分布的随机数来初始化。

第 22 行：将 W_4 保存到 model 属性中。

第 23 行：定义第 2 个卷积层神经元偏置值 b_4，由于本层采用 VALID 边缘填充方式，所以特征图尺寸将缩小为 10×10，因此偏置值 b_4 为[10, 10, 16]，即 16 个 10×10。

第 24 行：将 b_4 保存到 model 属性中。

第 25 行：通过卷积操作求出第 2 个卷积层的输入信号 z_4，卷积核为第 21 行定义的 W_4，宽度和高度方向的步长均为 1，边缘填充采用 VALID 模式（不填充），根据公式得出：

$$w_{\text{conv}} = \frac{w_{\text{in}} - F + 2P}{S} + 1 = \frac{14 - 5 + 2 \times 0}{1} + 1 = 10$$

最终得到的特征图尺寸为 10×10，共有 16 个这样的特征图。

第 26 行：将 z_4 保存到 model 属性中。

第 27 行：求 ReLU 神经元的输入信号 `a_4`。

第 28 行：将 `a_4` 保存到 `model` 属性中。

第 30 行：定义第 2 个最大池化层，池化窗口大小为 2×2 ，宽度和高度方向步长均为 2，边缘采用 2 像素 0 填充，根据公式得出：

$$W2 = \frac{W1 - F}{S} + 1 = \frac{10 - 2}{2} + 1 = 5$$

即最后得到的特征图为 16 个 5×5 的特征图。

第 31 行：将第 2 个最大池化层输出 `m_5` 保存到 `model` 属性中。

第 33 行：定义第 3 个卷积层的卷积核 `W_5`，卷积核尺寸为 5×5 ，输入信号 12 个通道，输出特征图数量为 120，采用均值为 0.0、标准差为 0.1 的正态分布的随机数来初始化。

第 34 行：将 `W_5` 保存到 `model` 属性中。

第 35 行：定义第 3 个卷积层神经元偏置值 `b_6`，由于本层得到的特征图尺寸为 1×1 ，共有 120 个特征图，所以 `b_6` 的形状为 $[1, 1, 120]$ 。

第 36 行：将 `b_6` 保存到 `model` 属性中。

第 37 行：定义卷积操作得到的输入信号 `z_6_raw`，采用第 33 行定义的卷积核，宽度和高度方向步长为 1，边缘填充采用 `VALID` 模式（不填充），根据公式得出：

$$w_{\text{conv}} = \frac{w_{\text{in}} - F + 2P}{S} + 1 = \frac{5 - 5 + 2 \times 0}{1} + 1 = 1$$

所以，最终特征图尺寸为 1，共有 120 个特征图。

第 38 行：将 `z_6_raw` 保存到 `model` 属性中。

第 39 行：由于这一层要接入后面的全连接层，因此需要将本层的特征图形状从 $[None, 1, 1, 120]$ 变为 $[None, 120]$ ，变为普通全连接层的形式，保存到 `z_6` 中。

第 40 行：将 `z_6` 保存到 `model` 属性中。

第 41 行：定义 ReLU 神经元输出 `a_6`。

第 42 行：将 `a_6` 保存到 `model` 属性中。

第 43 行：采用 Dropout 调整技术，保留 `keep_prob` 比例的神经元输出，其余神经元输出置为零，将经过 Dropout 的输出保存到 `a_6_dropout` 中。

第 44 行：将 `a_6_dropout` 保存到 `model` 属性中。

第 46 行：定义第 3 个卷积层到第 1 个全连接层的连接权值 `W_6`，形状为 120×84 ，用均值为 0.0、标准差为 0.1 的正态分布的随机数进行初始化。

第 47 行：将 `W_6` 保存到 `model` 属性中。

第 48 行：定义第 1 个卷积层的偏置值 `b_7`，其形状为 $[84]$ 。

第 49 行：将 `b_7` 保存到 `model` 属性中。

第 50 行：将经过 Dropout 的第 3 个卷积层的输出与连接权值相乘，并与偏置值相加，得到本层输入值 `z_7`。

第 51 行：将 `z_7` 保存到 `model` 属性中。

第 52 行：求出 ReLU 神经元输出 `a_7`。

第 53 行：将 `a_7` 保存到 `model` 属性中。

第 54 行：求出经过 Dropout 后本层的最终输出值 `a_7_dropout`。

第 55 行：将 `a_7_dropout` 保存到 `model` 属性中。

第 57 行：定义第 1 个全连接层到第二个全连接层，即输出层之间的连接权值 `W_7`，形状为`[84,10]`，并用均值为 0.0、标准差为 0.1 的正态分布的随机数进行初始化。

第 58 行：将 `W_7` 保存到 `model` 属性中。

第 59 行：定义第 2 个全连接层，即输出层偏置值 `b_8`，形状为`[10]`。

第 60 行：将 `b_8` 保存到 `model` 属性中。

第 61 行：求出第 2 个卷积层，即输出层的输入值 `z_8`。

第 62 行：将 `Z_8` 保存到 `model` 属性中。

第 63 行：求出 softmax 神经元的输出值 `y_`，即 10 种类别（0~9 的数字）所对应的类别。

第 64 行：将 `y_` 保存到 `model` 属性中。

第 66、67 行：定义输出标签计算值和正确值之间的交叉熵 `cross_entropy`。

第 68 行：将 `cross_entropy` 保存到 `model` 属性中。

第 69 行：可以选择采用 Adagrad 优化算法，注意在本例中没有使用这个优化算法。

第 70~74 行：定义代价函数为交叉熵再加上 L2 调整项（权值衰减）。

第 75 行：将代价函数 `loss` 保存到 `model` 属性中。

第 76~78 行：采用 Adam 优化算法，求代价函数最小值时的参数值。

第 79 行：将 `train_step` 保存到 `model` 属性中。

第 80 行：利用 TensorFlow 的 `argmax` 函数，分别求出计算类别向量每个样本的最大值下标和类别向量每个样本的最大值下标，并对其进行比较。

第 81 行：将 `correct_prediction` 保存到 `model` 属性中。

第 82 行：求出预测精度，首先调用 TensorFlow 的 `cast` 函数，将第 16 行的结果变为浮点数列表，格式为：`[1.0, 1.0, 0.0, 1.0, 1.0]`，这里假设只有 5 个样本。再调用 TensorFlow 的 `reduce_mean` 函数求出列表的平均值： $(1.0+1.0+0.0+1.0+1.0)/5=0.8$ ，这个值就是模型预测的精度。

第 83 行：将 `accuracy` 保存到 `model` 属性中。

第 84、85 行：返回模型定义的 `X`, `y_`, `y`, `keep_prob`, `cross_entropy`, `train_step`, `correct_prediction`, `accuracy`。

5.2.2 训练方法

讲解完模型构建方法之后，我们来看网络的训练方法，代码如下。

```
1 def train(self, mode=TRAIN_MODE_NEW, ckpt_file='work/lgr.ckpt'):
2     X_train, y_train, X_validation, y_validation, X_test, \
3         y_test, mnist = self.load_datasets()
4     X, y_, y, keep_prob, cross_entropy, train_step, correct_prediction, \
5         accuracy = self.build_model()
6     epochs = 10
```



```

7     saver = tf.train.Saver()
8     total_batch = int(mnist.train.num_examples/self.batch_size)
9     check_interval = 50
10    best_accuracy = -0.01
11    improve_threthold = 1.005
12    no_improve_steps = 0
13    max_no_improve_steps = 3000
14    is_early_stop = False
15    eval_runs = 0
16    eval_times = []
17    train_accs = []
18    validation_accs = []
19    with tf.Session() as sess:
20        sess.run(tf.global_variables_initializer())
21        if Cnn_Engine.TRAIN_MODE_CONTINUE == mode:
22            saver.restore(sess, ckpt_file)
23        for epoch in range(epochs):
24            if is_early_stop:
25                break
26            for batch_idx in range(total_batch):
27                if no_improve_steps >= max_no_improve_steps:
28                    is_early_stop = True
29                    break
30                X_mb_raw, y_mb = mnist.train.next_batch(self.batch_size)
31                X_mb = X_mb_raw.reshape([self.batch_size, 28, 28, 1])
32                sess.run(train_step, feed_dict={X: X_mb, y: y_mb,
33                    keep_prob: self.keep_prob})
34                no_improve_steps += 1
35                if batch_idx % check_interval == 0:
36                    eval_runs += 1
37                    eval_times.append(eval_runs)
38                    train_accuracy = sess.run(accuracy,
39                        feed_dict={X: X_train.reshape([-1, 28, 28, 1]),
40                            y: y_train, keep_prob: 1.0})
41                    train_accs.append(train_accuracy)
42                    validation_accuracy = sess.run(accuracy,
43                        feed_dict={X: X_validation.reshape([-1, 28, 28, 1]),
44                            y: y_validation,
45                                keep_prob: 1.0})
46                    validation_accs.append(validation_accuracy)
47                    if best_accuracy < validation_accuracy:
48                        if validation_accuracy / best_accuracy >= \
49                            improve_threthold:
50                            no_improve_steps = 0
51                            best_accuracy = validation_accuracy
52                            saver.save(sess, ckpt_file)
53                            print('{0}:{1}# train:{2}, validation:{3}'.format(
54                                epoch, batch_idx, train_accuracy,
55                                    validation_accuracy))
56            print(sess.run(accuracy, feed_dict={X: X_test.reshape([-1, 28, 28, 1]),
57                y: y_test, keep_prob: 1.0}))
58    plt.figure(1)
59    plt.subplot(111)
60    plt.plot(eval_times, train_accs, 'b-', label='train accuracy')
61    plt.plot(eval_times, validation_accs, 'r-',
62        label='validation accuracy')

```

```

63     plt.title('accuracy trend')
64     plt.legend(loc='lower right')
65     plt.show()

```

第 2、3 行：读入训练样本集输入信号集、训练样本集标签集、验证样本集输入信号集、验证样本集标签集、测试样本集输入信号集、测试样本集标签集。

第 4、5 行：创建模型，这里创建的模型是前面讲到的 ReLU 神经元模型。

第 6 行：循环学习整个训练样本集遍数。

第 7 行：初始化 TensorFlow 模型保存和恢复对象 saver。

第 8 行：用用户训练样本集中样本数除以迷你批次大小，得到迷你批次数量 total_batch。

第 9 行：每隔 50 次迷你批次学习，计算在验证样本集上的精度。

第 10 行：保存在验证样本集上所取得的最好的验证样本集精度。

第 11 行：定义验证样本集上精度提高 0.5% 时才算显著提高。

第 12 行：记录在验证样本集上精度没有显著提高学习迷你批次的次数。

第 13 行：在验证样本集精度最大值没有显著提高的情况下，允许学习迷你批次的数量。

第 14 行：是否终止学习过程。

第 15 行：评估验证样本集上识别精度的次数。

第 16 行：用 eval_times 列表保存评估次数，作为后面绘制识别精度趋势图的横坐标。

第 17 行：用 train_accs 列表保存在训练样本集上每次评估时的识别精度，作为后面图形深色曲线的纵坐标。

第 18 行：用 validation_accs 列表保存在验证样本集上每次评估时的识别精度，作为后面图形浅色曲线的纵坐标。

第 19 行：启动 TensorFlow 会话。

第 20 行：初始化全局参数。

第 21、22 行：如果模式为 TRAIN_MODE_CONTINUE，则读入以前保存的 ckpt 模型文件，初始化模型参数。

第 23 行：循环第 24~42 行操作，对整个训练样本集进行一次学习。

第 24、25 行：如果 is_early_stop 为真时，终止本层循环。

第 26 行：循环第 27~42 行操作，对一个迷你批次进行学习。

第 27~29 行：如果验证样本集上识别精度没有显著改善的迷你批次学习次数大于最大允许的验证样本集上识别精度没有显著改善的迷你批次学习次数，则将 is_early_stop 置为真，并退出本层循环。这会直接触发第 24、25 行的操作，终止外层循环，代表学习过程结束。

第 30 行：从训练样本集中取出一个迷你批次的输入信号集 X_mb_raw 和标签集 y_mb。

第 31 行：将训练样本集迷你批次设计矩阵 X_mb_raw 的形状变为 [batch_size, 28, 28, 1]，即变为 28×28 的图像，1 个通道。

第 32、33 行：调用 TensorFlow 计算模型输出、代价函数，求出代价函数对参数的导数，并应用梯度下降算法更新模型参数值。注意，此时采用 Dropout 调整技术，将隐藏层神经元保留比例作为一个参数 keep_prob 传给模型。

第 34 行：将验证样本集没有显著改善的迷你批次学习次数加 1。

第 35 行：如果连续进行了指定次数的迷你批次学习，则计算统计信息。

第 36 行：识别精度评估次数加 1。

第 37 行：将识别精度评估次数加入 `eval_times`（图形横坐标）列表中。

第 38~40 行：计算训练样本集上的识别精度。

第 41 行：将训练样本集上的识别精度加入训练样本集识别精度列表 `train_accs` 中。

第 42~45 行：计算验证样本集上的识别精度。注意，此时采用 Dropout 调整技术，将隐藏层神经元保留比例作为一个参数 `keep_prob` 传给模型，这里给出的是 1.0，即全部保留。这是一个惯例，即在训练时使用 Dropout 调整技术，在评估和运行时则不使用，读者一定要注意。

第 46 行：将验证样本集上的识别精度加入到验证样本集识别精度列表 `validation_accs` 中。

第 47 行：如果验证样本集上最佳识别精度小于当前验证样本集上的识别精度，执行第 48~52 行操作。

第 48~50 行：如果当前验证样本集上的识别精度比之前的最佳识别精度提高 0.5% 以上，则将验证样本集没有显著改善的迷你批次学习次数设为 0。

第 51 行：将验证样本集上最佳识别精度的值设置为当前验证样本集上识别精度的值。

第 52 行：将当前模型参数保存到 `ckpt` 模型文件中。

第 53~55 行：打印训练状态信息。

第 56、57 行：训练完成后，计算测试样本集上的识别精度，并打印出来。

第 58、59 行：初始化 `matplotlib` 绘图库。

第 60 行：绘制训练样本集上识别精度的变化趋势曲线，用蓝色绘制。

第 61、62 行：绘制验证样本集上识别精度的变化趋势曲线，用红色绘制。

第 63 行：设置图形标题。

第 64 行：在右下角添加图例。

第 65 行：具体绘制图像。

运行这个程序，就可以得到前文中 Sigmoid 及 ReLU 神经元模型的后台输出和识别精度曲线图了。

由图 5.24 可以看出，在训练样本集上的识别精度，与在训练过程中没有见过的验证样本集上的识别精度和测试样本集上的识别精度差别不大，但精度明显高于多层感知器模型，这说明卷积神经网络的识别精度还是非常高的。我们没有刻意优化网络结构，也没有调整超参数，如学习率、L2 调整项（权值衰减）、Early Stopping 标准和 Dropout 失活比例等。如果对这些参数进行调整，模型的识别精度还可以有很大的提升空间，有兴趣的读者可以进行尝试。

```
3:350# train:0.9843817949295044, validation:0.985599946594238
3:400# train:0.9836363792419434, validation:0.9833999872207642
3:450# train:0.9849091172218323, validation:0.9837999939918518
3:500# train:0.9816363453865051, validation:0.9801999926567078
4:0# train:0.9807817935943604, validation:0.9807999730110168
4:50# train:0.9868545532226562, validation:0.9872000217437744
4:100# train:0.9867454767227173, validation:0.9854000210762024
4:150# train:0.9853272438049316, validation:0.9843999743461609
4:200# train:0.9855636358261108, validation:0.9851999878883362
4:250# train:0.9809636473655701, validation:0.9825999736785889
4:300# train:0.9848909378051758, validation:0.984000027179718
4:350# train:0.9842908978462219, validation:0.984000027179718
4:400# train:0.9845454692840576, validation:0.9824000000953674
4:450# train:0.9873090982437134, validation:0.9861999750137329
4:500# train:0.9870363473892212, validation:0.9868000149726868
5:0# train:0.9865090847015381, validation:0.9850000143051147
5:50# train:0.9873636364936829, validation:0.9882000088691711
5:100# train:0.986181795970764, validation:0.98580002784729
5:150# train:0.9881636500358582, validation:0.9873999953269958
5:200# train:0.9845818281173706, validation:0.9832000136375427
5:250# train:0.9864727258682251, validation:0.9847999811172485
5:300# train:0.9857817888259888, validation:0.9869998885559082
5:350# train:0.9878363609313965, validation:0.9851999878883362
5:400# train:0.9856545329093933, validation:0.9851999878883362
5:450# train:0.9855999946594238, validation:0.9865999817848206
5:500# train:0.9879636168479919, validation:0.9869999885559082
6:0# train:0.986054539680481, validation:0.9850000143051147
0.9853
```

图 5.24 卷积神经网络训练结果

由图 5.25 可以看出，在训练样本集上的识别精度与在验证样本集上的识别精度基本重合，表明模型还没有出现过拟合的情况，说明我们的 Early Stopping 标准可能定得太高了，可以适当降低标准，识别精度还可以有较大的提升。另外，我们知道，曾取得图像识别竞赛冠军的卷积神经网络，尤其是残差网络（ResNet），足有 100 层之多，在有多 GPU 的机器上最少也要训练两三周，而这个网络，在普通没有 GPU 的机器上，运行时间也就是几分钟而已，表明我们还可以尝试向模型添加卷积层和最大池化层来提高网络的表现能力，再配合适当的调整技术，则可以得到更大的精度。另外，现在的网络卷积核采用的是 5×5 的尺寸，读者还可以将这个尺寸变为 3×3 的，看看结果是否有所不同。

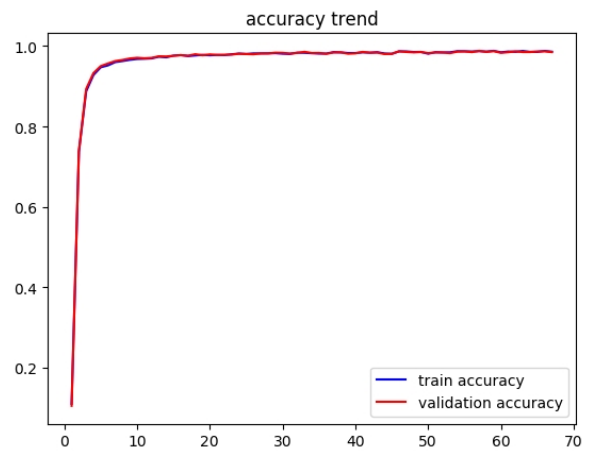


图 5.25 训练样本集和验证样本集上识别精度变化趋势

在卷积神经网络训练完成之后，就进入网络运行状态，对实际中遇到的全新样本进行预测，下面我们来看模型的运行方法。

5.2.3 运行方法

运行方法的代码如下：

```

1 def run(self, ckpt_file='work/lgr.ckpt'):
2     img_file = 'datasets/test6.png'
3     img = io.imread(img_file, as_grey=True)
4     raw = [1 if x<0.5 else 0 for x in img.reshape(784)]
5     #sample = np.array(raw)
6     X_train, y_train, X_validation, y_validation, \
7         X_test, y_test, mnist = self.load_datasets()
8     X, y_, y, keep_prob, cross_entropy, train_step, correct_prediction, \
9         accuracy = self.build_model()
10    sample = X_test[102]
11    X_run = sample.reshape(1, 28, 28, 1)
12    saver = tf.train.Saver()
13    digit = -1
14    with tf.Session() as sess:
15        sess.run(tf.global_variables_initializer())
16        saver.restore(sess, ckpt_file)
17        rst = sess.run(y_, feed_dict={X: X_run, keep_prob: 1.0})
18        print('rst:{0}'.format(rst))
19        max_prob = -0.1
20        for idx in range(10):
21            if max_prob < rst[0][idx]:
22                max_prob = rst[0][idx]
23                digit = idx
24        # W_1
25        W_1 = sess.run(self.model['W_1'], feed_dict={X: X_run, keep_prob: 1.0})
26        print('W_1:{0}'.format(W_1))
27        a_2 = sess.run(self.model['a_2'], feed_dict={X: X_run, keep_prob: 1.0})
28        fm1 = a_2[0, :, :, 0]
29        fm2 = a_2[0, :, :, 1]
30        fm3 = a_2[0, :, :, 2]
31        fm4 = a_2[0, :, :, 3]
32        fm5 = a_2[0, :, :, 4]
33        fm6 = a_2[0, :, :, 5]
34    img_in = sample.reshape(28, 28)
35    plt.figure(1)
36    plt.subplot(241)
37    plt.imshow(img_in, cmap='gray')
38    plt.title('result:{0}'.format(digit))
39    plt.axis('off')
40    # feature map1
41    plt.subplot(242)
42    plt.imshow(fm1, cmap='gray')
43    plt.axis('off')
44    plt.title('fm1')
45    # feature map2
46    plt.subplot(243)
47    plt.imshow(fm2, cmap='gray')
48    plt.axis('off')
49    plt.title('fm2')
50    # feature map3

```

```

51 plt.subplot(244)
52 plt.imshow(fm3, cmap='gray')
53 plt.axis('off')
54 plt.title('fm3')
55 # feature map4
56 plt.subplot(246)
57 plt.imshow(fm4, cmap='gray')
58 plt.axis('off')
59 plt.title('fm4')
60 # feature map5
61 plt.subplot(247)
62 plt.imshow(fm5, cmap='gray')
63 plt.axis('off')
64 plt.title('fm5')
65 # feature map6
66 plt.subplot(248)
67 plt.imshow(fm6, cmap='gray')
68 plt.axis('off')
69 plt.title('fm6')
70 plt.show()

```

第 2 行：用绘图软件做一个 28×28 的图像，在上面写一个数字，这里直接写一个印刷体的“5”。

第 3 行：以灰度图像方式读出图像内容。

第 4 行：先将其形状从二维 28×28 变为一维 784，然后根据每个像素的值进行处理：小于 0.5 取 1，否则取 0。

第 5 行：将其变为 numpy 数组，作为一个样本。

第 6、7 行：调用 load_datasets 方法，读入训练样本集、验证样本集、测试样本集的内容。

第 8、9 行：调用 build_model 方法，建立卷积神经网络模型。

第 10 行：取测试样本集中的第 103 个样本作为测试样本，我们的模型在训练过程中没有见到过测试样本集中的样本，因此可以模拟实际应用中遇到的情况。

第 11 行：将其变为[1, 28, 28, 1]的矩阵形式，可以称之为运行样本集，其中只有一个样本。

第 12 行：初始化 TensorFlow 的 saver 对象。

第 13 行：定义 digit 为最终识别出的 0~9 的数字，取-1 表示还没有识别结果。

第 14 行：启动 TensorFlow 会话来运行程序。

第 15 行：初始化变量。

第 16 行：恢复之前保存的模型参数文件，并初始化模型参数。

第 17 行：求以样本 X_run 为输入，在输出层经过 softmax 函数后的计算值，表示为每个类别的概率。注意：这里设置 Dropout 技术中隐藏层神经元的保留率为 1.0，即所有隐藏层神经元均参与运算，相当于多个随机网络共同投票得出最终结果。

第 18 行：打印出所有类别的概率值。

第 19 行：定义 max_prob 记录所有类别最大的概率值。

第 20 行：循环识别结果 rst 每个类别的概率。

第 21~23 行：如果最大概率小于当前类别的概率，将当前概率赋给最大概率，识别出的数字等于类别索引值，即为对应的 0~9 中的数字。当循环完所有类别后，就能找到概率最大的类别及其对应的数字了。

第 25 行：取出输入层到第 1 个卷积层的连接权值矩阵 W_1 （实际为其转置）。

第 26 行：打印出其内容。

第 27 行：求出第 1 个卷积层神经元输出值，即特征图。注意：这里设置 Dropout 技术中隐藏层神经元的保留率为 1.0，即所有隐藏层神经元均参与运算，相当于多个随机网络共同投票得出最终结果。

第 28 行：取出第 1 个特征图 $fm1$ 。

第 29 行：取出第 2 个特征图 $fm2$ 。

第 30 行：取出第 3 个特征图 $fm3$ 。

第 31 行：取出第 4 个特征图 $fm4$ 。

第 32 行：取出第 5 个特征图 $fm5$ 。

第 33 行：取出第 6 个特征图 $fm6$ 。

第 34 行：将模型输入信号重新变为 28×28 的黑白图像。

第 35 行：初始化 matplotlib 绘图库。

第 36~39 行：在 2 行 4 列的第 1 行第 1 列中绘制待识别图片。

第 40~44 行：在 2 行 4 列的第 1 行第 2 列中绘制特征图 1。

第 45~49 行：在 2 行 4 列的第 1 行第 3 列中绘制特征图 2。

第 50~54 行：在 2 行 4 列的第 1 行第 4 列中绘制特征图 3。

第 55~59 行：在 2 行 4 列的第 2 行第 2 列中绘制特征图 4。

第 60~64 行：在 2 行 4 列的第 2 行第 3 列中绘制特征图 5。

第 65~69 行：在 2 行 4 列的第 2 行第 4 列中绘制特征图 6。

第 70 行：同时显示三幅图像。

运行结果如图 5.26 所示。

```
[[[ 0.27460822  0.2049055  0.03427025  0.04984366  0.10102937 -0.17978966]]
 [[ 0.02901221  0.06168453 -0.03483592 -0.00340924  0.22166145 -0.02886526]]
 [[ 0.001467  0.12828436 -0.22415894 -0.12079416  0.26737645  0.1039087 ]]
 [[-0.10686503 -0.04714583 -0.41242868 -0.2023375  0.09262011  0.30800956]]
 [[-0.14217061  0.15219374 -0.359878  -0.06526492  0.0868542  0.14663348]]]
[[[ 0.26257625  0.26214835  0.13710046 -0.19534631  0.28979844 -0.32383344]]
 [[ 0.18766756  0.13829955  0.04700303 -0.23627278  0.26081216 -0.08565003]]
 [[-0.10610206  0.0725405  0.09258558 -0.13857083  0.09024068 -0.00759155]]
 [[-0.21432368  0.18178274 -0.09433907  0.04671084  0.11982096  0.29586279]]
 [[-0.16769278  0.21757986 -0.23243448  0.16815814 -0.20198356  0.1344438 ]]
 [[ 0.1713943  0.24376936  0.02930518 -0.05909403  0.17018753 -0.24636279]]
 [[ 0.21654491  0.24285294  0.25189462 -0.06036001  0.10425588 -0.17156938]]
 [[-0.05464235  0.3584241  0.2379331  0.04874515  0.06290098  0.00174851]]
 [[-0.26298586  0.21489124  0.27620515  0.18822894 -0.13213083  0.24888484]]
 [[-0.11865969  0.31510743  0.11779037  0.20735507 -0.27893052  0.18023042]]]
[[[ 0.23380457  0.13533913  0.04626744  0.2462295  0.23767291 -0.21327117]]
 [[ 0.15928821  0.05866562 -0.00071157  0.16886942  0.04069318 -0.08371975]]
 [[-0.06052112  0.02631545  0.22871238  0.27297142 -0.2092317  0.1332643 ]]
 [[-0.18039586  0.02614399  0.31675792  0.35568735 -0.24856195  0.1130347 ]]
 [[-0.03765234  0.15373316  0.30716178  0.07250497 -0.26044005  0.16669223]]]
[[[ 0.17606696 -0.21058582 -0.066641  0.06555062  0.0544896 -0.23050943]]
 [[ 0.10150024 -0.20023544 -0.08498108  0.1472065  0.05717795  0.01216208]]
 [[ 0.07570601 -0.3159222 -0.1529457  0.12197056 -0.15118395  0.14351602]]
 [[-0.03945933 -0.23990962 -0.08290513  0.14641857 -0.23936835  0.12509584]]
 [[-0.14574736 -0.22780475  0.02782533  0.05118438 -0.15782285  0.12591836]]]]]
```

图 5.26 第 1 个卷积层卷积核

上图中每一列代表一个 5×5 的卷积核，共有 6 个卷积核，用于生成第 1 个卷积层的 6 个特征图，如图 5.27 所示。

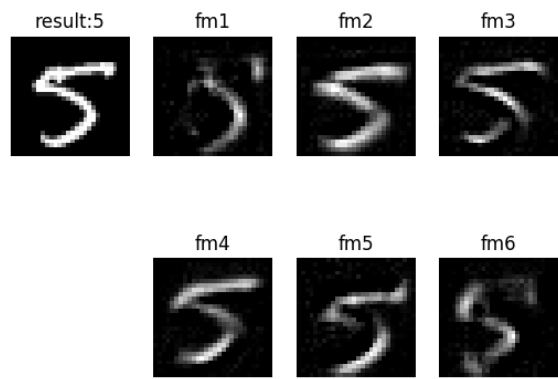


图 5.27 第 1 个卷积层上的 6 个特征图

本章讲述了卷积神经网络的基础知识，并且利用 TensorFlow 框架实现了卷积神经网络之父 Yann LeCun 提出的 LeNet5 网络，并将该网络用于 MNIST 手写数字识别数据集，通过综合应用当前主流的优化算法和调整技术，达到比较高的精度。卷积神经网络是当前图像处理领域应用最广泛的一种深度学习模型，在 ImageNet 竞赛中，近几年都是这种网络结构取得冠军，比较有代表性的网络有 Google 推出的 Inception v4 网络和微软推出的 ResNet 网络，还有将二者融合的 Inception-ResNet-v2，可以达到当前最高的识别精度。所有这些网络都可以找到原始论文，读者如果有比较好的机器，也可以尝试构造这些网络，并用 ImageNet 数据集进行训练。不过，ImageNet 数据集大约 120GB，训练这些网络通常需要以月来计。为了方便大家利用这些网络，一些牛人已经将这些模型搭建好了，并且将训练好的参数公开发布了，大家可以在此基础上做自己的实验。为了进一步降低应用门槛，TensorFlow 推出了一个新的子项目 tf-slim (<https://github.com/TensorFlow/models/tree/master/slim>)，这个项目里有当前主流的卷积神经模型，包括：LeNet5、AlexaNet、Inception 系列、ResNet 及 Inception-ResNet-v2。读者可以研究一下这个项目，看看利用它们怎样实现这些网络，通过学习这些代码，可以极大地提高大家的应用水平。

第 6 章

递归神经网络

到目前为止，我们所讨论的神经网络都是前馈神经网络，即网络中没有回路，信号是单向传播的。如果我们允许网络中出现回路，就可以得到递归神经网络。因为允许回路的存在，递归神经网络可以视为最深的神经网络，同时也是最灵活、表现能力最强的一种神经网络。以模式分类问题为例，可以分为三种类型的问题：如果一个输入严格地对应一个输出结果，属于普通模式识别问题，称为序列分类（Sequence Classification），则可以使用我们前面所述的神经网络来实现；如果多个输入对应一个输出，甚至多个输入对应多个输出，就必须采用递归神经网络进行处理了，如果每个输出的位置是固定的，则称这类问题为区块分类（Segment Classification）；如果无法确定输入界限，则称这类问题为时序分类（Temporal Classification）。递归神经网络主要用来处理后两类问题，例如连续手写文字识别（包括在线和离线两种方式，不是识别单个字母，而是识别一句话或一段文字）和语音识别等领域。

在这一章中，先给大家介绍一下递归神经网络的基本原理，并用 Python 完成一个简单的递归神经网络。我们将用这个神经网络来预测字符序列的下一个字符，并且向大家展示一个典型应用——神经网络写作系统。接着会讨论将递归神经网络用于图像处理，这是近年来的一个热点研究方向，通常与卷积神经网络结合使用，用于图像分割和图像标记（Image Caption），本章就以图像标记为例，讲述递归神经网络在图像中的应用。

6.1 递归神经网络原理

递归神经网络是允许有环形结构的神经网络，具有很多种类型，如 Elman 网络、Jordan 网络、时延神经网络、Echo State 网络等。如果在网络运行时需要将来的信息，例如在手写

文字识别中可能根据这个字后续的字来帮助网络做出正确的判断，可以组成双向递归神经网络。为了解决 Segment Labeling 等问题的长短时记忆网络，以及在图像识别应用中多维层级递归神经网络，还有目前比较流行的神经图灵机等。由于递归神经网络允许回路的存在，可以表示解决问题的上下文，因此可以解决更加复杂的问题，目前主要的应用领域包括语音识别、自然语言处理（NLP）、图像标注等多个领域。

在本节中，只讨论具有单个自连接隐藏层的递归神经网络。虽然这种神经网络看似很简单，尤其是只需要上百行 Python 代码即可实现，但是其功能却很强大，可以通过学习写出像模像样的文章来。

6.1.1 递归神经网络表示方法

只具有一个隐藏层的递归神经网络结构如图 6.1 所示。

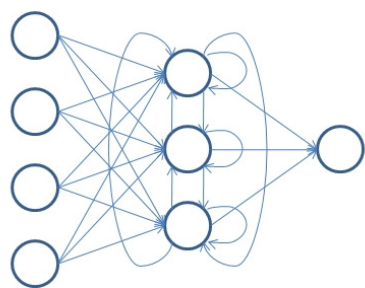


图 6.1 递归神经网络结构图

与多层感知器模型相比，递归神经网络隐藏层不仅与输出层连接，而且隐藏层节点之间具有自连接，即隐藏层的输出不仅会传输给输出层，而且还会传输给隐藏层自身。因为按照图 6.1 表示递归神经网络不直观，所以在实际应用中通常用递归神经网络的展开形式来表示，如图 6.2 所示。

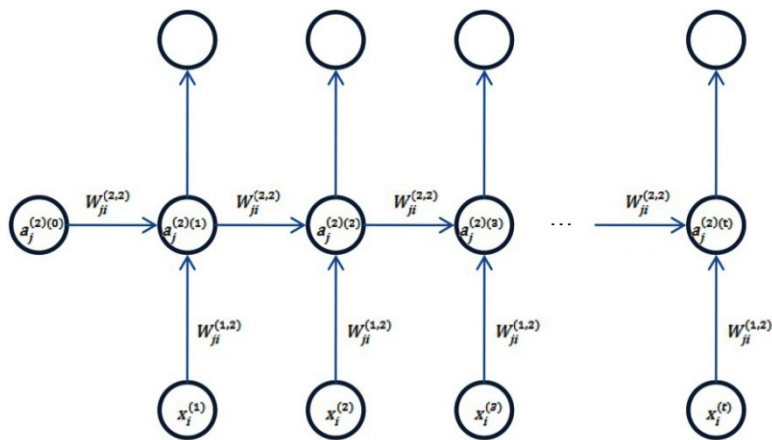


图 6.2 递归神经网络的展开形式

图中显示的是输出层第 i 个神经元与隐藏层第 j 个神经元相连接的情况，在 $t=1$ 时刻，输入层输出为 $x_i^{(1)}$ ， i 代表神经元节点序号， (1) 代表第 1 个样本，隐藏层第 j 个神经元不仅与输入层第 i 个神经元相连，还与假想的 $t=0$ 时刻的隐藏层神经元节点相连，此时隐藏层第 j 个节点在 $t=1$ 时刻的输入为前述两个神经元输出信号的线性组合。通常认为 $t=0$ 时刻神经元输出为 0（当然也可以取随机值，但是取零比较简单），通过激活函数就可以求出 $t=1$ 时刻隐藏层第 j 个神经元的输出，这时的输出不仅输出给输出层，同时也输出给隐藏层自身，依次类推。

6.1.2 数学原理

递归神经网络前向传播过程与多层感知器模型相似，根据图 6.2 递归神经网络的展开形式，在 t 时刻隐藏层第 j 个神经元的输入为：

$$z_j^{(2)(t)} = \sum_{i=1}^{n_1} W_{ji}^{(1,2)} x_i^{(t)} + \sum_{k=1}^{n_2} W_{jk}^{(2,2)} a_k^{(2)(t-1)} \quad (1)$$

式中各字母的意义与多层感知器模型基本相同， z 表示神经元输入信号， j 表示第 j 个神经元， (2) 表示第 2 层， (t) 表示在 t 时刻的值， $t \in \{1, 2, \dots, T\}$ 。 n_1 为第 1 层神经元数量， W 代表连接权值， ji 代表由输入层第 i 个神经元指向隐藏层第 j 个神经元， $(1,2)$ 代表从第 1 层指向第 2 层的连接权值， x 代表输入向量第 i 个分量的第 t 个样本，这里样本的序号就是 t 时刻的值。

等式右边第 2 项是隐藏层输入到隐藏层时产生的项， n_2 为隐藏层数量， $(2,2)$ 表明由第 2 层指向第 2 层， jk 表示第 k 个隐藏层神经元输出又接入到第 j 个隐藏层神经元的输入。 a 的下标 k 代表第 k 个神经元，上标 (2) 代表第 2 层，上标 $(t-1)$ 代表在 $t-1$ 时刻该神经元的输出。一般规定 $t=0$ 时刻隐藏层神经元输出为 0。

有了隐藏层第 j 个神经元在 t 时刻的输入之后，就可以求出该神经元在 t 时刻的输出：

$$a_j^{(2)(t)} = f_2(z_j^{(2)(t)}) \quad (2)$$

对于序列标记和块标记，我们可以使用与多层感知器模型类似的算法。

输出层神经元的输入值计算与隐藏层相同：

$$z_1^{(L)(t)} = \sum_{j=1}^{n_{L-1}} W_{1j}^{(L-1,L)} a_j^{(L-1)(t)} \quad (3)$$

由于输出层只有一个神经元，其激活函数为 Sigmoid，我们用 σ 来表示，其输出为：

$$a_1^{(L)(t)} = f_o(z_1^{(L)(t)}) = \sigma(z_1^{(L)(t)}) \quad (4)$$

1. 两类别判断

因为是二分类问题，因此输出层神经元的输出可以理解为 $c=1$ 时的概率。因为只能有两种类别可以选择，因此输出层的概率分布为伯努利分布： $\text{Bernoulli}(\emptyset) = \emptyset$ ，用 c 来表示输出值所属类别： $c \in \{0,1\}$ ，则有如下公式：

$$p(c = 1 | \mathbf{a}^{(L-1)(t)}; \mathbf{W}^{(L-1,L)}) = a_1^{(L)(t)} = f_L(z_1^{(L)(t)}) \quad (5)$$

$$p(c = 0 | \mathbf{a}^{(L-1)(t)}; \mathbf{W}^{(L-1,L)}) = 1 - a_1^{(L)(t)} = 1 - f_L(z_1^{(L)(t)}) \quad (6)$$

式中， $\mathbf{a}^{(L-1)(t)}$ 为最后隐藏层的输出向量，且 $\mathbf{a}^{(L-1)(t)} \in \mathbb{R}^{n_{L-1}+1}$ ， $\mathbf{W}^{(L-1,L)}$ 为最后隐藏层到输出层的连接权值矩阵。

将上述两式进行合并得：

$$p(c | \mathbf{a}^{(L-1)(t)}; \mathbf{W}^{(L-1,L)}) = (a_1^{(L)(t)})^c (1 - a_1^{(L)(t)})^{1-c} \quad (7)$$

当 $c=1$ 时，等号右边第二项指数为 0，其值为 1，就变为式 (5)。当 $c=0$ 时，等号右边第一项指数为 0，其值为 1，就变为式 (6)。

将负对数似然函数定义为代价函数（损失函数），则：

$$\begin{aligned} \mathcal{L}(\mathbf{a}^{(L-1)(t)}, c) &= -\log(p(c | \mathbf{a}^{(L-1)(t)}; \mathbf{W}^{(L-1,L)})) \\ &= ((1-c)\log(1 - a_1^{(L)(t)}) - c\log(a_1^{(L)(t)})) \end{aligned} \quad (8)$$

我们的问题就转化为通过调整递归神经网络参数，使负对数似然函数最小的问题，可以通过梯度下降算法来达到这一目的。

现在很多教科书在讲多层感知器模型学习算法时，是将输出层的最小平方误差：

$(\text{MSE} = \|\mathbf{y}^{(s)} - \mathbf{a}_1^{(L)}\|^2)$ 作为代价函数，通过调整连接权值，使其达到最小的。

但是直接利用最小平方误差对连接权值进行求导，应用梯度下降算法调整权值时，由于输出层神经元的激活函数为 Sigmoid 函数，当输入值过大或过小时，会出现饱和现象，此时梯度会非常小，造成学习效率很低，这也在一定程度上造成了多层感知器模型在十年前逐渐让位于支撑向量机（SVM）。

当我们采用负对数似然函数作为代价函数，对连接权值进行求导时，应用梯度下降算法调整权值可以有效地避免这一问题，可以视为多层感知器模型在过去 10~20 年间最重要的进展之一，另一进展是隐藏层采用 ReLU 神经元，而不是 Sigmoid 或双曲正切神经元。这是由于算法层面的这些进展，才使古老的 BP 算法在当前深度学习时代，重新得到研究人员的重视。

若求负对数似然函数对于连接权值的偏导数，可以采用链式求导法则，先求负对数似然函数对输出层输出的导数：

$$\begin{aligned}
 \frac{\partial \mathcal{L}(\mathbf{a}^{(L-1)(t)}, c)}{\partial a_1^{(L)(t)}} &= \frac{\partial}{\partial a_1^{(L)(t)}} \left((1-c) \log(1 - a_1^{(L)(t)}) - c \log(a_1^{(L)(t)}) \right) \\
 &= \frac{1-c}{1 - a_1^{(L)(t)}} - \frac{c}{a_1^{(L)(t)}} = \frac{(1-c)a_1^{(L)(t)} - c(1 - a_1^{(L)(t)})}{a_1^{(L)(t)}(1 - a_1^{(L)(t)})} \\
 &= \frac{a_1^{(L)(t)} - c}{a_1^{(L)(t)}(1 - a_1^{(L)(t)})} \quad (9)
 \end{aligned}$$

接下来求输出层输出对最后隐藏层输出的导数：

$$\begin{aligned}
 \delta_1^{(L)(t)} &= \frac{\partial \mathcal{L}(\mathbf{a}^{(L-1)(t)}, c)}{\partial z_1^{(L)(t)}} = \frac{\partial \mathcal{L}(\mathbf{a}^{(L-1)(t)}, c)}{\partial a_1^{(L)(t)}} \cdot \frac{\partial a_1^{(L)(t)}}{\partial z_1^{(L)(t)}} \\
 &= \frac{a_1^{(L)(t)} - c}{a_1^{(L)(t)}(1 - a_1^{(L)(t)})} \cdot \frac{\partial f_L(z_1^{(L)(t)})}{\partial z_1^{(L)(t)}} = \frac{a_1^{(L)(t)} - c}{a_1^{(L)(t)}(1 - a_1^{(L)(t)})} (a_1^{(L)(t)}(1 - a_1^{(L)(t)})) \\
 &= a_1^{(L)(t)} - c \quad (10)
 \end{aligned}$$

我们注意到，在式（10）中， $c \in \{0,1\}$ ，其实际就是训练样本 $(\mathbf{x}^{(s)}, \mathbf{y}^{(s)})$ 中的 $\mathbf{y}^{(s)}$ ，这也是在简单网络示例中，输出层误差直接用 $a_1^{(L)} - \mathbf{y}^{(s)}$ 来求的原因。

由此可见，式（10）中定义的就是输出层的误差，有了误差之后，我们就可以运行误差反向传播算法了。

下面我们先讨论在输出层不止有一个神经元的情况下，怎样求出输出层误差，然后再统一讲解误差反向传播和权值调整问题。

2. 多类别判断

如果处理的问题类别 $K \geq 3$ 时，输出层神经元数目就为 $n_L = K$ ，此时输出层的分布将为多项式分布，输出层的输出为：

$$p(c^{(t)} = i | \mathbf{a}^{(L-1)(t)}; \mathbf{W}^{(L-1,L)}) = a_i^{(L)(t)} = \frac{e^{z_i^{(L)(t)}}}{\sum_{k=1}^K e^{z_k^{(L)(t)}}} \quad (11)$$

可以视为每个类别出现的概率，网络的输出取所有输出层神经元输出值最大的一个，输出值为 1，其余神经元输出值为 0，例如当 $n_L = 5$ 时，第 2 个类别式（11）所对应的值最大，即其出现的概率最大，用向量 $\mathbf{c}^{(t)} = [0 \ 1 \ 0 \ 0 \ 0]^T$ 表示，则可以定义为：

$$p(\mathbf{c}^{(t)} | \mathbf{a}^{(L-1)(t)}; \mathbf{W}^{(L-1,L)}) = \prod_{k=1}^{n_L} \left(a_k^{(L)(t)} \right)^{c_k^{(t)}} \quad (12)$$

负对数似然函数可以定义为：

$$\mathcal{L}(\mathbf{a}^{(L-1)(t)}, \mathbf{c}^{(t)}) = -\log \left(\prod_{k=1}^{n_L} \left(a_k^{(L)(t)} \right)^{c_k^{(t)}} \right) = -\sum_{k=1}^{n_L} c_k^{(t)} \log \left(a_k^{(L)(t)} \right) \quad (13)$$

接下来的任务是通过调整连接权值，使负对数似然函数达到最小，从而最终使多层感知器模型输出误差值达到最小。所以需要对上式求偏导，由于直接对连接权值求导比较困难，故应用链式求导法则，首先对输出层输出求偏导，由于输出层由多个神经元组成，下面对第 i 个神经元求偏导：

$$\frac{\partial \mathcal{L}(\mathbf{a}^{(L-1)(t)}, \mathbf{c}^{(t)})}{\partial a_i^{(L)(t)}} = -\frac{c_i^{(t)}}{a_i^{(L)(t)}}$$

下面求负对数似然函数对输出层输入量的偏导：

$$\frac{\partial \mathcal{L}(\mathbf{a}^{(L-1)(t)}, \mathbf{c}^{(t)})}{\partial z_k^{(L)(t)}} = \sum_{i=1}^{n_L} \frac{\partial \mathcal{L}(\mathbf{a}^{(L-1)(t)}, \mathbf{c}^{(t)})}{\partial a_i^{(L)(t)}} \cdot \frac{\partial a_i^{(L)(t)}}{\partial z_i^{(L)(t)}} = \sum_{i=1}^{n_L} \left(-\frac{c_i^{(t)}}{a_i^{(L)(t)}} \right) \cdot \frac{\partial a_i^{(L)(t)}}{\partial z_i^{(L)(t)}}$$

经过推导可得：

$$\delta_i^{(L)(t)} = \frac{\partial \mathcal{L}(\mathbf{a}^{(L-1)(t)}, \mathbf{c}^{(t)})}{\partial z_i^{(L)(t)}} = a_i^{(L)(t)} - c_i^{(t)}$$

即输出层每个神经元误差的计算公式。

3. 误差反向传播

递归神经网络误差反向传播主要有两种算法：RTRL 和 BPTT。在这里我们将介绍 BPTT 算法，因为 BPTT 算法在概念上更简单一些，而且计算效率较高，同时也是实际应用中较常见的一种算法。

在上节的讨论中，我们对输出层误差的计算，实际上与之前介绍的多层感知器模型完全相同，我们重点需要解决的就是隐藏层误差的计算。

根据递归神经网络的拓扑结构，隐藏层的误差也是由两部分组成，第一部分是由输出层误差反向传播得到的，第二部分是由 $t+1$ 时刻隐藏层误差反向传播而来的。

所以为了计算反向传播误差，我们假设时间序列的上限（训练样本数）为 m ，则需要从 m 时刻开始计算误差反向传播。这时假设 $m+1$ 时刻隐藏层反向传播过来的误差为 0，因为 $m+1$ 时刻没有输入输出，因此没有误差。

根据上述假设，在第 t 时刻，我们设输出层神经元的输入信号为 zf ，隐藏层神经元的输入信号为 zr （由下一时刻隐藏层输出产生），隐藏层第 i 个节点接收到的误差为：

$$\begin{aligned}
 \delta_i^{(2)(t)} &= \frac{\partial \mathcal{L}(\mathbf{a}^{(L-1)(t)}, \mathbf{c}^{(t)})}{\partial z f_i^{(2)(t)}} + \frac{\partial \mathcal{L}(\mathbf{a}^{(L-1)(t)}, \mathbf{c}^{(t)})}{z r_i^{(2)(t)}} \quad ① \\
 &= \frac{\partial \mathcal{L}(\mathbf{a}^{(L-1)(t)}, \mathbf{c}^{(t)})}{\partial a f_i^{(2)(t)}} \frac{\partial a f_i^{(2)(t)}}{\partial z f_i^{(2)(t)}} + \frac{\partial \mathcal{L}(\mathbf{a}^{(L-1)(t+1)}, \mathbf{c}^{(t+1)})}{\partial a r_i^{(2)(t+1)}} \frac{\partial a r_i^{(2)(t+1)}}{\partial z r_i^{(2)(t+1)}} \quad ② \\
 &= \frac{\partial a f_i^{(2)(t)}}{\partial z f_i^{(2)(t)}} \sum_{k=1}^{n_L} \frac{\partial \mathcal{L}(\mathbf{a}^{(L-1)(t)}, \mathbf{c}^{(t)})}{\partial z f_k^{(L)(t)}} \frac{\partial z f_k^{(L)(t)}}{\partial a f_i^{(2)(t)}} + \frac{\partial a r_i^{(2)(t+1)}}{\partial z r_i^{(2)(t+1)}} \sum_{j=1}^{n_2} \frac{\partial \mathcal{L}(\mathbf{a}^{(L-1)(t+1)}, \mathbf{c}^{(t+1)})}{\partial z r_j^{(2)(t+1)}} \frac{\partial z r_j^{(2)(t+1)}}{\partial a r_i^{(2)(t+1)}} \quad ③ \\
 &= f_2'(z f_i^{(2)(t)}) \sum_{k=1}^{n_L} \delta_k^{(L)(t)} W_{ki}^{(L-1,L)} + f_2'(\partial z r_i^{(2)(t+1)}) \sum_{j=1}^{n_2} \delta_j^{(2)(t+1)} W_{ij}^{(2,2)} \quad (14)
 \end{aligned}$$

在①处第一项为输出层转播到隐藏层，第二项为隐藏层 $t+1$ 时刻传输出隐藏层。

对于权值调整，首先求代价函数对权值的偏导：

$$\frac{\partial \mathcal{L}}{\partial w_{ji}} = \sum_{t=1}^m \frac{\partial \mathcal{L}}{\partial z_j^{(t)}} \frac{\partial z_j^{(t)}}{\partial w_{ji}} = \sum_{t=1}^m \delta_j^{(t)} a_i^{(t)} \quad (15)$$

有了上述公式，就可以进行权值调整，规则与多层感知器模型相同。

4. 权值调整

求出每个神经元的误差之后，需要根据误差对权值进行调整，我们首先要求出负对数似然函数相对于该神经元连接权值的导数。

根据定义，第 1 层第 i 个神经元的误差为：

$$\frac{\partial \mathcal{L}}{\partial w_{ji}} = \sum_{t=1}^m \frac{\partial \mathcal{L}}{\partial z_j^{(t)}} \frac{\partial z_j^{(t)}}{\partial w_{ji}} = \sum_{t=1}^m \delta_j^{(t)} a_i^{(t)}$$

求出每个神经元对连接权值的导数之后，我们就可以根据梯度下降算法调整该神经元相关的连接权值了。

我们定义在第 r 轮的连接权值调整中，权值调整量为：

$$\Delta W_{ij}^{(l)(r)} = -\alpha \frac{\partial \mathcal{L}}{\partial W_{ij}^{(l)(r)}}$$

式中， α 为学习率，控制多层感知器学习算法收敛速度，通常 $\alpha \in (0,1)$ ，例如 $\alpha = 0.1$ 。每调整一次权值，算作一轮， r 的值加 1。

直接使用梯度下降算法容易陷入局部最小值点，因为在局部最小值点，导数的值为零，权值将不再进行调整，因此也就无法走出局部最小值点了。

为了克服上述缺点，人们在调整权值时引入了动量项，其权值调整量为：

$$\Delta W_{ij}^{(l)(r)} = m\Delta W_{ij}^{(l)(r-1)} - \alpha \frac{\partial \mathcal{L}}{\partial W_{ij}^{(l)(r)}}$$

式中， m 为动量项，且 $m \in [0,1]$ ，用其乘以上一次权值调整的值作为权值调整的惯性，来协助算法逃出局部极小值点。

6.1.3 简单递归神经网络应用示例

在讲述了递归神经网络基本原理之后，我们来介绍一个简单递归神经网络的典型应用。在这个应用中，我们将利用递归神经网络学习一系列文本数据，然后利用该网络预测下一个单词，最终使这个简单的递归神经网络可以写出类似的文本来。其实这就是计算机写作的基本原理，利用该技术可以让计算机写出文章。虽然递归神经网络确实是计算机写作的基础，但是离真正写出专业的文章还差得很远，不仅包括训练数据准备，还包括其他方方面面的内容，有兴趣的读者可以在这个网络的基础上，参考其他文献自行进行研究。

首先需要有一个文本文件，为递归神经网络提供训练样本，文件内容如下：

其实这就是计算机写作的基本原理，利用该技术，是可以让计算机写出文章的。但是正像利用黑火药可以制造出枪炮，但是掌握黑火药技术离制造枪炮还有很远的距离。同理，我们当前递归神经网络确实是计算机写作的基础，但是离真正写出专业的文章，还差得很远很远，不仅包括训练数据准备，还包括其他方方面面的内容，读者有兴趣可以在这个网络的基础上，参考其他文献，自己进行研究。

读入文本文件后，建立一个单词索引，代码如下：

```
1 import numpy as np
2
3 data = open('input.txt', 'r').read()
4 chars = list(set(data))
5 data_size, vocab_size = len(data), len(chars)
6 print('d=%d; v=%d' % (data_size, vocab_size))
7 char_to_ix = { ch:i for i,ch in enumerate(chars) } #dict
8 ix_to_char = { i:ch for i,ch in enumerate(chars) } #dict
9
```

第3行：打开 input.txt 文件，将内容读入到 data 变量。

第4行：去除 data 中重复的汉字，将其生成列表赋给 chars 变量。

第5行：取出总字数（汉字可能重复）和单词数。

第7行：建立通过汉字找到索引值的字典变量 char_to_ix，其格式如下：

```
{ '造': 0, '就': 39, '离': 2, '容': 42, '让': 4, '该': 43, ' ': 44, '距': 87, '>
经': 1, '专': 47, '用': 46, '药': 5, '归': 6, '理': 40, '们': 48, '掌': 49, '备'
: 74, '训': 50, '正': 66, '自': 7, '但': 51, '准': 8, '章': 9, '方': 52, '实': 4
1, '还': 10, ' ': 54, '业': 12, '利': 55, '趣': 13, '础': 56, '制': 15, '其': 5
7, '握': 58, '他': 17, '仅': 59, '括': 3, '献': 14, '参': 60, '机': 20, '这': 21
, '在': 61, '兴': 62, '数': 63, '出': 22, '基': 64, '技': 65, '炮': 23, '面': 18
, '网': 67, '作': 68, '得': 24, '像': 69, '己': 70, '确': 71, '个': 75, '练': 73
, '术': 25, '的': 26, '内': 76, '很': 19, '火': 77, '究': 78, '原': 27, '络': 94
, '有': 72, '枪': 28, '可': 79, '前': 45, '研': 53, '当': 80, '计': 29, '远': 16
, '考': 30, '包': 83, '同': 82, '算': 81, '真': 31, '差': 84, '以': 85, '本': 86
```



```
, '递': 32, '神': 33, '行': 88, '文': 89, '进': 34, '者': 35, '\n': 36, '我': 91, '读': 92, '是': 93, '黑': 90, '写': 11, '不': 37, '据': 38}
```

第 8 行：建立通过索引值查找到汉字的字典变量 `ix_to_char`，其格式如下：

```
{0: '造', 1: '经', 2: '离', 3: '括', 4: '让', 5: '药', 6: '归', 7: '自', 8: '准', 9: '章', 10: '还', 11: '写', 12: '业', 13: '趣', 14: '献', 15: '制', 16: '远', 17: '他', 18: '面', 19: '很', 20: '机', 21: '这', 22: '出', 23: '炮', 24: '得', 25: '术', 26: '的', 27: '原', 28: '枪', 29: '计', 30: '考', 31: '真', 32: '递', 33: '神', 34: '进', 35: '者', 36: '\n', 37: '不', 38: '据', 39: '就', 40: '理', 41: '实', 42: '容', 43: '该', 44: ' ', 45: '前', 46: '用', 47: '专', 48: '们', 49: '掌', 50: '训', 51: '但', 52: '方', 53: '研', 54: ' ', 55: '利', 56: '础', 57: '其', 58: '握', 59: '仅', 60: '参', 61: '在', 62: '兴', 63: '数', 64: '基', 65: '技', 66: '正', 67: '网', 68: '作', 69: '像', 70: '己', 71: '确', 72: '有', 73: '练', 74: '备', 75: '个', 76: '内', 77: '火', 78: '究', 79: '可', 80: '当', 81: '算', 82: '同', 83: '包', 84: '差', 85: '以', 86: '本', 87: '距', 88: '行', 89: '文', 90: '黑', 91: '我', 92: '读', 93: '是', 94: '络'}
```

下面来定义递归神经网络结构，代码如下：

```
10 hidden_size = 100
11 seq_length = 25
12 learning_rate = 1e-1
13
14 Wxh = np.random.randn(hidden_size, vocab_size)*0.01
15 Whh = np.random.randn(hidden_size, hidden_size)*0.01
16 Why = np.random.randn(vocab_size, hidden_size)*0.01
17 bh = np.zeros((hidden_size, 1))
18 by = np.zeros((vocab_size, 1))
19
20
```

第 10 行：定义隐藏层神经元数量。

第 11 行：定义递归神经网络中序列的最大长度。这实际上是普通递归神经网络的一个缺陷，即无法处理序列的长度，只能以超参数的形式出现。对于这个问题，我们将通过下一章中讨论的长短时记忆网络来加以解决。

第 12 行：定义学习率。

第 14 行：定义输入层到隐藏层的连接权值矩阵，其中输入层神经元数量为语料库中不同汉字的数量。

第 15 行：定义隐藏层与隐藏层神经元节点间的连接权值矩阵。

第 16 行：定义隐藏层到输出层间的连接权值矩阵，其中输出层神经元数量为语料库中不同汉字的数量。

第 17 行：隐藏层神经元的偏移量。

第 18 行：输出层神经元的偏移量。

下面我们来看代价函数的计算，代码如下：

```
21 def lossFun(inputs, targets, hprev):
22     xs, hs, ys, ps = {}, {}, {}, {}
23     hs[-1] = np.copy(hprev)
24     loss = 0
25     for t in range(len(inputs)):
26         xs[t] = np.zeros((vocab_size, 1))
27         xs[t][inputs[t]] = 1
28         hs[t] = np.tanh(np.dot(Wxh, xs[t]) + np.dot(Whh, hs[t-1]) + bh)
29         ys[t] = np.dot(Why, hs[t]) + by
30         ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t]))
31         loss += -np.log(ps[t][targets[t], 0])
32     dWxh, dWhh, dWhy = np.zeros_like(Wxh), np.zeros_like(Whh), \
33                         np.zeros_like(Why)
34     dbh, dby = np.zeros_like(bh), np.zeros_like(by)
35     dhnext = np.zeros_like(hs[0])
36     for t in reversed(range(len(inputs))):
```

```

37     dy = np.copy(ps[t])
38     dy[target[t]] -= 1
39     dwhy += np.dot(dy, hs[t].T)
40     dby += dy
41     dh = np.dot(why.T, dy) + dhnext
42     dhraw = (1 - hs[t] * hs[t])*dh
43     dwxh += np.dot(dhraw, xs[t].T)
44     dwhh += np.dot(dhraw, hs[t-1].T)
45     dhnext = np.dot(whh.T, dhraw)
46     for dparam in [dwxh, dwhh, dwhy, dbh, dby]:
47         np.clip(dparam, -5, 5, out=dparam)
48     return loss, dwxh, dwhh, dwhy, dbh, dby, hs[len(inputs)-1]
49

```

第 21 行：定义代价函数计算函数。

- **inputs**: 给定的输入序列。
- **targets**: 希望的输出序列。
- **hprev**: 前一时刻隐藏层的输出值。

例如，根据输入文件，在网络刚开始运行时，**inputs** 的值为：

```
[57, 41, 21, 39, 93, 29, 81, 20, 11, 68, 26, 64, 86, 27, 40, 44, 55, 46, 43, 65,
25, 44, 93, 79, 85]
```

将其转换为汉字为：“其实这就是计算机写作的基本原理，利用该技术，是可以。”

我们希望网络预测的序列 **targets** 为：

```
[41, 21, 39, 93, 29, 81, 20, 11, 68, 26, 64, 86, 27, 40, 44, 55, 46, 43, 65, 25,
44, 93, 79, 85, 4]
```

转换为汉字为：“实这就是计算机写作的基本原理，利用该技术，是可以让。”

网络的学习过程为：首先给网络“其”字，让其预测下一个字，因为网络通过学习知道“实”字在“其”字后面的概率最大，因此将“实”字作为下一个字进行预测。然后依此类推，预测出其后 24 个汉字，并与 **targets** 中给出的正确答案进行比对，如果不正确就进行权值和偏移量的调整。这就是网络的基本学习过程。

第 22 行：定义字典类型变量。**xs** 表示输入，**hs** 表示隐藏层，**ys** 表示输出层，**ps** 表示输出层概率。

第 23 行：将参数中隐藏层上一时刻的初始值赋给 **hs[-1]**。

第 24 行：代价函数的初始值为 0。

第 25 行：对所有输入信号进行循环，由变量 **seq_length** 可知，输入信号元素数为 25。每个输入元素为一个时刻，故共有 25 个时刻。

第 26 行：求出当前时刻的输入信号，输入字典 **key** 为当前时刻，值为维度为单字数的列表，初始值为零。

第 27 行：将 **inputs** 中本时刻在 **xs** 字典列表中对应的值置为 1。还以第一次运行为例，此时 **t=0**，初始化 **xs[0]=[0, 0, ..., 0]** 共有单个汉字维，在本例中为 95。而 **inputs[0]=57**，查上面索引号到汉字表可得该字为“其”，则 **x[0]=[0, 0, ..., 1, ..., 0]**，其中为 1 的项为第 57 项，即下标为 56 的那一项。

第 28 行：有了输入信号，上一时刻隐藏层状态后，可以按以下公式计算隐藏层当前状态值：

$$\begin{aligned}
 a_j^{(2)(t)} &= \sum_{i=1}^{n_1} W_{ji}^{(1,2)} x_i^{(t)} + \sum_{i=1}^{n_2} W_{ji}^{(2,2)} a_j^{(2)(t-1)} + b_j^{(2)(t)} \\
 &= \sum_{i=1}^{n_1} Wxh_{ji}xs[t][i] + \sum_{i=1}^{n_2} Whh_{ji}hs[-1][i] + bh[j]
 \end{aligned}$$

将上式用 Numpy 表示，即可得到程序中的代码。

第 29 行：定义输出字典在键为 t 时刻的输出向量，公式如下：

$$\mathbf{y} = W_{hy}\mathbf{hs} + \mathbf{yb}$$

式中， \mathbf{y} , \mathbf{hs} , \mathbf{yb} 为向量， W_{hy} 为隐藏层到输出层的连接权值矩阵。

第 30 行：计算输出层神经元所对应汉字的出现概率，公式为：

$$p_i^{(t)} = \frac{e^{W_{hy}h_i + b_i}}{\sum_{j=1}^K e^{W_{hy}h_j + b_j}}$$

将上式转换为 Numpy 表示，即可得到程序中的代码。

第 31 行：代价函数为负对数似然函数，求其值。

第 32、33 行：定义连接权值导数矩阵，分别存放输入层到隐藏层、隐藏层到隐藏层、隐藏层到输出层连接权值，元素初始值为 0。

第 34 行：定义 dbh 和 $db\mathbf{y}$ 保存隐藏层、输出层偏移量导数向量。

第 35 行：定义隐藏层输出值的导数。

第 36 行：定义反向传播循环，从后向前循环，即从序列长度 25 一直循环到 0。

第 37、38 行：求输出层导数 dy ，首先将本时刻输出层概率复制到 dy 向量中，然后在正确输出汉字的相应神经元的导数上减 1。

第 39 行：求隐藏层到输出层连接权值导数，输出层导数向量与隐藏层输出值向量的点积，并求其累积和。公式推导如下：

$$\begin{aligned}
 dy &= \frac{\partial y}{\partial z} \\
 \frac{\partial y}{\partial w} &= \frac{\partial y}{\partial z} \frac{\partial z}{\partial w} = dy \cdot \mathbf{hs}
 \end{aligned}$$

在上面的推导中，为了简化起见，我们没有使用正式的符号，第 37、38 行求出的 dy 实际上表示输出层输出对输出层输入的导数，而我们要求输出层输出与隐藏层到输出层权值的导数时，需要应用链式求导法则，先求输出层输出对输出层输入的导数，再求输出层输入对隐藏层到输出层连接权值的导数。上面的第一项就是 dy ，而输出层输出对隐藏层到输出层连接权值的导数，正好等于隐藏层的输出值。

第 40 行：输出层偏移量的导数为输出层输出对输出层输入导数累加和。

第 41 行：求输出层输出对隐藏层输出的导数 dh ，公式推导如下：

$$\frac{\partial y}{\partial h} = \frac{\partial y}{\partial z} \frac{\partial z}{\partial h} = dy \cdot W_{hy}$$

首先应用链式求导法则求输出层输出对输出层输入的导数，再求输出层输入对隐藏层输出的导数。其中第一部分即为我们第 37、38 行求出的 dy ，而输出层输入对隐藏层输出的导数为隐藏层到输出层的连接权值。

第 42 行：求输出层输出对隐藏层输入的导数 dh_{raw} ，公式推导如下：

$$\frac{\partial y}{\partial z^{(2)}} = \frac{\partial y}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} = dy \cdot W_{hy} \cdot (1 - a^{(2)} \cdot a^{(2)})$$

式中，上标(3)代表输出层，(2)代表隐藏层， z 代表输入， a 代表输出。

因为我们在隐藏层使用的是双曲正切为激活函数的神经元，而双曲正切函数的导数为：

$$\tanh(x) = 1 - (\tanh(x))^2$$

第 43 行：求输出层输出对输入层到隐藏层连接权值导数，公式推导如下：

$$\frac{\partial y}{\partial W_{xh}} = \frac{\partial y}{\partial z_y} \frac{\partial z_y}{\partial a_h} \frac{\partial a_h}{\partial z_h} \frac{\partial z_h}{\partial W_{xh}} = dh_{raw} \cdot x$$

第 44 行：求输出层输出对隐藏层到隐藏层连接权值导数，公式推导如下：

$$\frac{\partial y}{\partial W_{hh}} = \frac{\partial y}{\partial z_y} \frac{\partial z_y}{\partial a_h} \frac{\partial a_h}{\partial z_h} \frac{\partial z_h}{\partial W_{hh}} = dh_{raw} \cdot h_{prev} = dh_{prev} \cdot hs[t-1]$$

由于隐藏层输入是由两部分组成：输入层输出和上一时刻隐藏层输出。第 43、44 行分别求输出层输出对输入层到隐藏层连接权值及隐藏层到隐藏层连接权值的导数。

第 45 行：求输出层输出对上一时刻隐藏层输出的导数，公式推导如下：

$$\frac{\partial y}{\partial a_h^{(t-1)}} = \frac{\partial y}{\partial z_y} \frac{\partial z_y}{\partial a_h} \frac{\partial a_h}{\partial z_h} \frac{\partial z_h}{\partial a_h^{(t-1)}} = dh_{prev} \cdot W_{hh}$$

根据链式求导法则，隐藏层 t 时刻输出对隐藏层 $t-1$ 时刻输出的导数，正好是隐藏层到隐藏层的连接矩阵。

第 46、47 行：利用数值方法，保证上述所求导数的正确性。

第 48 行：返回代价函数值，所求参数的导数，当前时刻隐藏层的输出值。

在网络训练完成之后，需要让网络自动生成一段文字，这一功能由 `sample` 函数来实现，代码如下：

```
50 def sample(h, seed_ix, n):
51     x = np.zeros((vocab_size, 1))
52     x[seed_ix] = 1
53     ixes = []
54     for t in range(n):
55         h = np.tanh(np.dot(Wxh, x) + np.dot(Whh, h) + bh)
56         y = np.dot(Wyh, h) + by
57         p = np.exp(y) / np.sum(np.exp(y))
58         ix = np.random.choice(range(vocab_size), p=p.ravel())
59         x = np.zeros((vocab_size, 1))
60         x[ix] = 1
61         ixes.append(ix)
```

```

62     return ixes
63
64

```

第 50 行：定义采样函数。

❑ **h**：隐藏层输出值向量，初始传入时全为 0，代表上一时刻隐藏层输出值向量。

❑ **seed_ix**：以哪个字开始。

❑ **n**：共产生多少个汉字的文章。

第 51 行：生成一个维度为所有汉字的向量，其元素值为 0。

第 52 行：用 **seed_ix** 设置第一个汉字对应的索引值为 1。

第 53 行：生成空列表，用于存储生成汉字的索引值。

第 54 行：循环生成 **n** 个汉字序列。

第 55 行：根据输入信号和上一时刻隐藏层输出值，求出当前隐藏层输出。

第 56 行：求出输出层的输出。

第 57 行：计算输出层各个汉字的概率。

第 58 行：取出输出层神经元，即所有汉字概率最大一项的索引值。

第 59 行：生成一个新的全 0 的输入信号。

第 60 行：将刚才求出概率最大的索引值对应项的值置为 1，即将下一个汉字对应索引值置为 1。

第 61 行：将刚生成汉字的索引加入到最终生成汉字索引的列表中。

程序的主循环如下：

```

65 n, p = 0, 0
66 mWxh, mWwh, mWhy = np.zeros_like(Wxh), np.zeros_like(Wwh), \
67     np.zeros_like(Why)
68 mbh, mby = np.zeros_like(bh), np.zeros_like(by)
69 smooth_loss = -np.log(1.0/vocab_size)*seq_length
70 while True:
71     if p+seq_length+1 >= len(data) or n == 0:
72         hprev = np.zeros((hidden_size, 1))
73         p = 0
74     inputs = [char_to_ix[ch] for ch in data[p:p+seq_length]]
75     targets = [char_to_ix[ch] for ch in data[p+1:p+seq_length+1]]
76     if n % 100 == 0:
77         sample_ix = sample(hprev, inputs[0], 200)
78         txt = ''.join(ix_to_char[ix] for ix in sample_ix)
79         print('-----\n %s \n-----' % (txt))
80     loss, dwxh, dwhh, dwhy, dbh, dby, hprev = lossFun(inputs, targets, \
81         hprev)
82     smooth_loss = smooth_loss * 0.999 + loss * 0.001
83     if n % 100 == 0:
84         print('iter %d, loss: %f' % (n, smooth_loss))
85     for param, dparam, mem in zip([Wxh, Wwh, Why, bh, by], \
86         [dwxh, dwhh, dwhy, dbh, dby], \
87         [mWxh, mWwh, mWhy, mbh, mby]):
88         mem += dparam * dparam
89         param += -learning_rate * dparam / np.sqrt(mem + 1e-8)
90     p += seq_length
91     n += 1

```

运行上面的程序，可以输出与下面类似的结果：

这其实就是计算机写作的基本原理，利用该技术，是可以让计算机写出文章的。但是正像利用黑火药可以制造出枪炮，但是掌握黑火药技术离制造枪炮还有很远的距离。同理，我们当前递归神经网络确实是计算机写作的基础，但是离真正写出专业的文章，还差得很远很远，不仅包括训练数据准备，还包括其他方方面面的内容，读者有兴趣可以让计算机写出文章的。但是正像利用黑火药可以制造出枪炮，但是掌握黑火药技术离制造枪炮还有很远的距离。

同

```
-----
iter 8700, loss: 0.070950
-----
```

离真正写出专业的文章，还差得很远很远，不仅包括训练数据准备，还包括其他方方面面的内容，读者有兴趣可以让计算机写出文章的。但是正像利用黑火药可以制造出枪炮，但是掌握黑火药技术离制造枪炮还有很远的距离。同理，我们当前递归神经网络确实是计算机写作的基础，但是离真正写出专业的文章，还差得很远很远，不仅包括训练数据准备，还包括其他方方面面的内容，读者有兴趣可以让计算机写出文章的。但是正像利用黑火药可以制造出枪炮，但是掌握黑火药技术离制造枪炮还有很远的距离。同理，我们当前递归神经网络确实是计算机写作的基础，但是离真正写出专业的文章，还差得很远很远，不仅包括训练

```
-----
iter 8800, loss: 0.067994
-----
```

，但是掌握黑火药技术离制造枪炮还有很远的距离。同理，我们当前递归神经网络确实是计算机写作的基础，但是离真正写出专业的文章，还差得很远很远，不仅包括训练数据准备，还包括其他方方面面的内容，读者有兴趣可以让计算机写出文章的。但是正像利用黑火药可以制造出枪炮，但是掌握黑火药技术离制造枪炮还有很远的距离。同理，我们当前递归神经网络确实是计算机写作的基础，但是离真正写出专业的文章，还差得很远很远，不仅包括训练

由上面的运行结果可以看出，我们这个只有 90 多行的递归神经网络，基本上也能写出语句大体通顺的文字，可见递归神经网络的威力。

第 65 行：定义 n 为迭代次数， p 为字符串指针位置。

第 66、67 行：对输入层到隐藏层、隐藏层到隐藏层、隐藏层到输出层的连接权值矩阵，定义内存辅助矩阵，主要用于保证数值计算的精确性。

第 68 行：对隐藏层偏移量、输出层偏移量，定义内存辅助变量，主要用于保证数值计算的精确性。

第 69 行：定义变量平滑代价函数的计算值。

第 70 行：定义无限循环，用于进行网络训练。

第 71 行：如果是第一次运行，或者字符串位置指针 p 超出了语料的范围，则执行 72~73 行程序，进行初始化。

第 72 行：定义 $t=-1$ 时刻隐藏层输出值全为 0。

第 73 行：字符串位置指针 p 指向训练语料库的开头。

第 74 行：从语料库中字符串位置指针 p 开始连续取序列长度（seq_length=25）个汉字，并求出其索引值，保存到 inputs 中。

第 75 行：从语料库中字符串位置指针 $p+1$ 开始连续取序列长度（seq_length=25）个汉字，并求出其索引值，保存到 targets 中。

第 76~79 行：每训练迭代 100 次，通过 sample 函数生成一段文字的索引值，将索引值变为对应的汉字，并打印到屏幕上。

第 80 行：调用 lossFun 函数，求出代价函数值，网络参数的导数，隐藏层当前时刻的输出值。

第 82~84 行：对代价函数值进行数值计算方法处理，并每隔 100 次迭代，打印经过平滑后的代价函数值。

第 85~89 行：对参数进行更新，为了使更新操作更精确，这里使用了数值计算技巧，对于每个参数：首先求出参数的平方，再将参数的平方加上一个特别小的数并开方，将平方值与开方值相除，得到该参数的计算值。参数调整的方法为：原参数值-学习率×参数计算值。

第 90 行：字符串位置指针 p 向后移动序列长度（seq_length=25）位。

第 91 行：迭代次数 n 的值加 1。

以上就是一个最简的递归神经网络的程序实现，除了个别数值计算的技巧，全部是 Numpy 的基本操作，但是却可以实现比较复杂的功能。

上面的代码只是网上的一段代码，非常适合用来理解基本概念。但是这段程序实际上并没有区分培训阶段和运行阶段，也没有培训结束控制，培训结果也没有进行持久化保存，所以如果想要拿来使用，还需对这个程序进行一些改造。

需要添加以下内容。

`rnn_initialize` 函数：从模型文件中读入参数值，初始化网络。

`train` 函数：利用语料库进行训练，可以设置代价函数值结束条件和迭代次数结束条件。当培训结束后，可以将模型参数保存到文件中。

`run` 函数：从模型文件中读入参数值，初始化网络，生成指定长度的文字。

由于有了上面的基础，这些功能的实现相对来讲就很简单了，读者可以自己尝试实现这些功能，开发一个递归神经网络工具类。

在 <https://github.com/yt7589/dlp.git> 项目的 `book/chp06/rnn_demo.py` 是本示例的完整代码，读者可以参考一下。读者可以替换 `input.txt` 文件的内容，长度要适中，几百至几千字即可，大约训练几分钟，这个例子就可以写出比较像样的段落了。

6.2 图像标记

所谓图像标记，就是给出一张图片，让机器用一段话或几个单词描述图片中的内容。随着 Web 2.0 的发展，用户产生内容（UGC）正在以指数级增长，其中 50% 以上是图像数据。目前我们可以相对高效地对文字信息进行处理（全文检索技术），但几乎没有任何办法来处理图像。所以有人形容图像信息就是 Internet 中的暗物质，虽然数量巨大，但是却不为我们所用。如果机器可以自动给图像打上标记，那么我们就可以像处理文字一样处理图像信息，这将产生巨大的应用价值。图像标记就是要完成这一任务的技术。

6.2.1 建立开发环境

在这一节中，我们将使用在 2014 年发布的微软 COCO 数据集，这个数据集是图像标记事实上的标准数据集。这个数据集包括 80000 张训练图片和 40000 张验证图片，每张图片有 5 个标签，是由亚马逊员工进行人工标记的。

对于这些数据，我们首先通过卷积神经网络 AlexNet 进行训练，然后将 FC7 的特征作为图片的特征，经过主成分分析算法，得到 512 维输入向量。将这些特征输入到递归神经网络中，用 COCO 数据集进行训练。训练样本集中同样保存了图片的原始 URL，可以随时查看原始图片，以检验算法的效果。

下面以斯坦福大学 cs231n 计算机视觉课中第三个大作业的图像标注作业为例，来讲解图像标注应用。在介绍具体程序之前，我们先来看一下 cs231n 图像标注作业的基本情况。在 cs231n 中，使用了 ipython notebook 开发环境，可以通过交互式方式，测试驱动开发方式，完成递归神经网络搭建，实现图像标注功能。但是由于 cs231n 课程对递归神经网络部分的讲解非常少，程序中只给出需要实现的函数，说明特别简略，因此这部分对想自学这门课程的人来说，还是相当复杂的。同时，这部分代码是基于 Python 2.7.x 系统来实现的，而本书使用的一直是 Python 3.5 系统，因此需要将原来的代码升级到 Python 3.5 系统。在这里，我们将 ipython notebook 代码变为了普通的 Python 代码，同时将 Python 2.7 系统代码升级到 Python 3.5 系统，最后完整实现了递归神经网络相关代码。

首先下载 cs231n 作业 3: http://cs231n.stanford.edu/winter1516_assignment3.zip，下载后解压到 a3 目录。

启动 Python 虚拟环境，需要配置开发环境，将所有依赖库保存到 requirements.txt 文件中，可以用 pip 来统一进行安装：

```
pip install -r requirements.txt
```

下面我们来下载图像标记训练样本集，进入到 cs231n/datasets 目录，运行脚本命令来下载相应的训练样本集：

```
./get_coco_captions.sh
./get_tiny_imagenet_a.sh
./get_pretrained_model.sh
```

由于我们需要高效率的 CNN、RNN 执行代码，因此需要 cython 库，进入 cs231n 目录，运行如下命令：

```
python setup.py build_ext --inplace
```

经过上述步骤，我们就建立了一个 RNN 网络的开发环境，就可以正式进行开发了。由于 RNN 图像标记应用采用测试驱动开发（TDD）方式，我们将采取先写测试用例，然后编写代码通过测试用例的方式，一步步实现图像标记功能。

6.2.2 图像标记数据集处理

先创建 rnn_imgcap.py 文件，载入微软 coco 库图像标注训练样本集：

```
1 import time, os, json
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 from cs231n.gradient_check import eval_numerical_gradient, \
6     eval_numerical_gradient_array
7 from cs231n.rnn_layers import *
8 from cs231n.captioning_solver import CaptioningSolver
9 from cs231n.classifiers.rnn import CaptioningRNN
10 from cs231n.coco_utils import load_coco_data, sample_coco_minibatch, \
11     decode_captions
12 from cs231n.image_utils import image_from_url
13
14 plt.rcParams['figure.figsize']=(10.0, 8.0)
15 plt.rcParams['image.interpolation'] = 'nearest'
16 plt.rcParams['image.cmap'] = 'gray'
17
18 def rel_error(x, y):
```



```

19     return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x)+np.abs(y))))
20
21 def load_data_ric():
22     data = load_coco_data(pca_features=True)
23     for k,v in data.items():
24         if type(v) == np.ndarray:
25             print('%s, %s, %s, %s' % (k, type(v), v.shape, v.dtype))
26         else:
27             print('%s, %s, %d' % (k, type(v), len(v)))
28     return data
29

```

第 1~12 行：引入所需要的库文件，其中有一些文件是 cs231n 课程中的源文件，但是已经移到 Python 3.5.x 版本。

第 14~16 行：初始化 matplotlib 库，用于绘制图形和图像。

第 18、19 行：求出两个向量或矩阵的差，用于验证算法的正确性。

第 21 行：定义 load_data_ric 函数，用于取出图像标注训练数据。

第 22 行：读入微软 coco 库图像标注训练样本集。在该样本集中，图像特性由 AlexNet 的 FC7 层取出，因为这里只演示图像标注，所以没有使用 AlexNet 上的原始图像特征，而是经过主程序分析，选择了 512 个特征作为图像特征。

第 23 行：我们取出的 data 是一个字典类型变量，所以打印出了键值对信息。

运行这段程序：

```

341 if __name__ == '__main__':
342     print('RNN Image Caption Application')
343     data = load_data_ric()

```

运行结果为：

```

RNN Image Caption Application
train_features, <class 'numpy.ndarray'>, (82783, 512), float32
val_image_idxes, <class 'numpy.ndarray'>, (195954,), int32
word_to_idx, <class 'dict'>, 1004
val_features, <class 'numpy.ndarray'>, (40504, 512), float32
train_urls, <class 'numpy.ndarray'>, (82783,), <U63
val_urls, <class 'numpy.ndarray'>, (40504,), <U63
val_captions, <class 'numpy.ndarray'>, (195954, 17), int32
train_captions, <class 'numpy.ndarray'>, (400135, 17), int32
idx_to_word, <class 'list'>, 1004
train_image_idxes, <class 'numpy.ndarray'>, (400135,), int32

```

下面我们来看一下数据集中的数据，代码如下：

```

30 def show_data_image_ric(data):
31     batch_size = 3
32     captions, features, urls = sample_coco_minibatch(data, \
33                                                     batch_size=batch_size)
34     for i, (caption, url) in enumerate(zip(captions, urls)):
35         plt.imshow(image_from_url(url))
36         plt.axis('off')
37         cation_str = decode_captions(caption, data['idx_to_word'])
38         plt.title(cation_str)
39         plt.show()
40

```

程序入口：

```

342 if __name__ == '__main__':
343     print('RNN Image Caption Application')
344     data = load_data_ric()
345     show_data_image_ric(data)

```

第 31 行：指定迷你批次大小为 3，即想显示 3 个图像及其标注信息。

第 32 行：调用 sample_coco_minibatch 方法，从训练样本集中取出 3 条记录，分别为图

像标注、图像特征、图像 URL。

第 34 行：对每个样本进行循环。

第 35 行：从样本中的 URL 取出网络中的图片并显示。

第 36 行：由于我们显示图像，所以不画坐标轴。

第 37 行：将标记中单词索引转换成单词。

第 38 行：将图像标注单词作为图像标题。

第 39 行：显示图像和标题。

程序运行效果如图 6.3 所示。

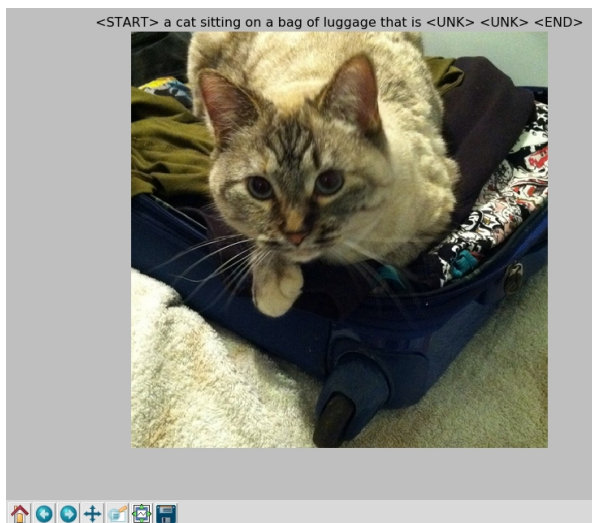


图 6.3 微软图像标注数据集 coco 示例

我们看到图像是一只猫，标注为一只猫坐在行李箱上。程序一共会显示 3 张这样的图片。

6.2.3 单步前向传播

下面我们来看递归神经网络的实现，首先是一个训练样本在网络的前向传播过程。为了开发这一功能，先来定义一下测试用例，代码如下：

```
41 def test_rnn_step_forward():
42     N, D, H = 3, 10, 4
43     x = np.linspace(-0.4, 0.7, num=N*D).reshape(N, D)
44     prev_h = np.linspace(-0.2, 0.5, num=N*H).reshape(N, H)
45     Wx = np.linspace(-0.1, 0.9, num=D*H).reshape(D, H)
46     Wh = np.linspace(-0.3, 0.7, num=H*H).reshape(H, H)
47     b = np.linspace(-0.2, 0.4, num=H)
48     next_h, _ = rnn_step_forward(x, prev_h, Wx, Wh, b)
49     print('next_h=%s' % next_h)
50     expected_next_h = np.asarray([
51         [-0.58172089, -0.50182032, -0.41232771, -0.31410098],
52         [ 0.66854692,  0.79562378,  0.87755553,  0.92795967],
53         [ 0.97934501,  0.99144213,  0.99646691,  0.99854353]]
54     )
55     print('next_h error:%s' % format(rel_error(expected_next_h, next_h), '.8e'))
56
```

第 41 行：定义递归神经网络单个样本前向传播程序的测试用例。

第 42 行：N 为迷你样本数量，D 为输入信号维度，H 为隐藏层神经元数量。

第 43 行：将-0.4~0.7 分为 $N \times D = 3 \times 10 = 30$ 段，将这些点组成数组，维度为二维数组，即 3 个样本，每个样本为 10 维。

第 44 行：定义上一时刻隐藏层输出。将-0.2~0.5 分为 $N \times H = 3 \times 4 = 12$ 段，将这些点组成二维数组，即 3 个样本，每个样本为 4 维。

第 45 行：定义输入层到隐藏层连接权值矩阵。将-0.1~0.9 分为 $D \times H = 10 \times 4 = 40$ 个点，将这些点组成 10 行 4 列的矩阵。

第 46 行：定义隐藏层到隐藏层连接权值矩阵。将-0.3~0.7 分为 $H \times H = 4 \times 4 = 16$ 个点，将这些点组成 4 行 4 列的矩阵。

第 47 行：定义隐藏层偏移值。将-0.2~0.4 分为 $H = 4$ 个点，将这些点组成 4 维向量。

第 48 行：调用 `rnn_step_forward` 函数，求出当前状态隐藏层输出值。

第 49 行：显示计算出的结果。

第 50~54 行：定义按照前面定义的输入，正确的隐藏层输出值。

第 55 行：计算计算出的隐藏层输出值与正确的隐藏层输出值之间的误差值，这里采用科学计数法显示，有效位数为 8。

下面我们来开发递归神经网络单步前向传播功能，打开 `cs231n/rnn_layers.py` 文件，代码如下：

```
1 import numpy as np
2
3 def rnn_step_forward(x, prev_h, Wx, Wh, b):
4     """
5     递归神经网络（RNN）单步前向传播功能，采用tanh为激活函数。输入信号维度
6     为D，隐藏层神经元数量为H，迷你批次的大小为N（包含N个样本）。
7     参数：
8     - x：时刻t的输入信号，为二维数组（迷你批次大小N，维度D）
9     - prev_h：上一时刻隐藏层输出值，维度（迷你批次大小N，隐藏层神经元数量H）
10    - Wx：输入层到隐藏层的连接权值矩阵，维度（输入信号维度D，隐藏层神经元数量H）
11    - Wh：隐藏层到隐藏层的连接权值，维度（隐藏层神经元数量H，隐藏层神经元数量H）
12    - b：隐藏层偏移量，维度（隐藏层神经元数量H）
13    - b：Biases of shape (H,)
14
15    返回值为元组：
16    - next_h：本时刻隐藏层输出值，维度（迷你批次大小N，隐藏层神经元数量H）
17    - cache：缓存值，包括输入信号x，输入层到隐藏层连接权值矩阵Wx，隐藏层到
18              隐藏层连接权值矩阵Wh，上一时刻隐藏层输出值prev_h，本时刻隐藏层
19              输出值next_h，隐藏层偏移量b
20    """
21    next_h, cache = None, None
22    next_h = np.tanh(np.dot(x, Wx) + np.dot(prev_h, Wh) + b)
23    cache = (x, Wx, Wh, prev_h, next_h, b)
24    return next_h, cache
```

第 22 行：计算当前时刻隐藏层输出，公式为：

$$\text{next_h}_i = \text{np.tanh}(W_{xh}x + W_{hh} \cdot \text{prev_h} + b)$$

第 23 行：将输入信号 x 、输入层到隐藏层连接权值矩阵 W_x 、隐藏层到隐藏层连接权值矩阵 W_h 、上一时刻隐藏层输出值 prev_h 、本时刻隐藏层输出值 next_h 、隐藏层偏移量 b 加入到缓存中。

第 24 行：返回当前状态隐藏层输出值和缓存的变量。

测试递归神经网络单步前向传播函数，代码如下：

```
342 if __name__ == '__main__':
343     print('RNN Image Caption Application')
344     data = load_data_ric()
345     test_rnn_step_forward()
```

运行结果为：

```
next_h=[[-0.58172089 -0.50182032 -0.41232771 -0.31410098]
 [ 0.66854692  0.79562378  0.87755553  0.92795967]
 [ 0.97934501  0.99144213  0.99646691  0.99854353]]
next_h error:6.29242143e-09
```

由上面的打印结果可以看出，计算出来的值与我们认为的正确值相差很小，可以认为这个实现是正确的。

6.2.4 单步反向传播

在完成了递归神经网络单步前向传播后，我们来看怎样进行单步反向传播。首先写出测试用例，代码如下：

```
57 def test_rnn_step_backward():
58     N, D, H = 4, 5, 6
59     x = np.random.randn(N, D)
60     h = np.random.randn(N, H)
61     Wx = np.random.randn(D, H)
62     Wh = np.random.randn(H, H)
63     b = np.random.randn(H)
64     out, cache = rnn_step_forward(x, h, Wx, Wh, b)
65     dnext_h = np.random.randn(*out.shape)
66     fx = lambda x: rnn_step_forward(x, h, Wx, Wh, b)[0]
67     fh = lambda prev_h: rnn_step_forward(x, h, Wx, Wh, b)[0]
68     fWx = lambda Wx: rnn_step_forward(x, h, Wx, Wh, b)[0]
69     fWh = lambda Wh: rnn_step_forward(x, h, Wx, Wh, b)[0]
70     fb = lambda b: rnn_step_forward(x, h, Wx, Wh, b)[0]
71     dx_num = eval_numerical_gradient_array(fx, x, dnext_h)
72     dprev_h_num = eval_numerical_gradient_array(fh, h, dnext_h)
73     dWx_num = eval_numerical_gradient_array(fWx, Wx, dnext_h)
74     dWh_num = eval_numerical_gradient_array(fWh, Wh, dnext_h)
75     db_num = eval_numerical_gradient_array(fb, b, dnext_h)
76
77     dx, dprev_h, dWx, dWh, db = rnn_step_backward(dnext_h, cache)
78
79     print('dx error:%s' % format(rel_error(dx_num, dx), '.8e'))
80     print('dprev_h error:%s' % format(rel_error(dprev_h_num, dprev_h), '.8e'))
81     print('dWx error:%s' % format(rel_error(dWx_num, dWx), '.8e'))
82     print('dWh error:%s' % format(rel_error(dWh_num, dWh), '.8e'))
83     print('db error:%s' % format(rel_error(db_num, db), '.8e'))
84
```

第 58 行：定义迷你批次大小 N （包含样本个数），输入信号维度 D ，隐藏层神经元数 H 。

第 59 行：定义输入信号为（迷你批次大小 N ，输入信号维度 D ）的二维数组，初始值为随机数。

第 60 行：上一时刻隐藏层输出值 h 为（迷你批次大小 N ，隐藏层神经元数量 H ），所有元素值为随机数。

第 61 行：输入层到隐藏层连接权值矩阵 W_x 为（输入信号维度 D ，隐藏层神经元数量 H ），所有元素值为随机数。

第 62 行：隐藏层到隐藏层连接权值矩阵 W_h 为（隐藏层神经元数量 H ，隐藏层神经元

数量 H)，所有元素值为随机数。

第 63 行：定义隐藏层偏移量 b 为（隐藏层神经元数量 H ），所有元素值为随机数。

第 64 行：通过调用 `rnn_step_forward` 函数，求出当前隐藏层输出值 `out`。

第 65 行：定义输出层的误差反向传播到隐藏层得到的误差 `dnext_h`。

第 66~75 行：采用数值计算方法求出对各参数的导数：对输入信号的导数 `dx_num`，对上一时刻隐藏层输出的导数 `dprev_h_num`，对输入层到隐藏层连接权值的导数 `dWx_num`，对隐藏层到隐藏层连接权值的导数 `dWh_num`，对隐藏层偏移量的导数 `db_num`。

第 77 行：通过调用 `rnn_step_backward` 函数，求出对输入信号的导数 `dx`、对上一时刻隐藏层输出的导数 `dprev_h`、对输入层到隐藏层连接权值的导数 `dWx`、对隐藏层到隐藏层连接权值的导数 `dWh`、对隐藏层偏移量的导数 `db`。

第 79 行：求出数值计算与通过 `rnn_step_backward` 求出的对输入信号导数的误差，并以 8 位有效数字的科学计数法显示。

第 80 行：求出数值计算与通过 `rnn_step_backward` 求出的对上一时刻隐藏层输出值导数的误差，并以 8 位有效数字的科学计数法显示。

第 81 行：求出数值计算与通过 `rnn_step_backward` 求出的对输入层到隐藏层连接权值导数的误差，并以 8 位有效数字的科学计数法显示。

第 82 行：求出数值计算与通过 `rnn_step_backward` 求出的对隐藏层到隐藏层连接权值导数的误差，并以 8 位有效数字的科学计数法显示。

第 83 行：求出数值计算与通过 `rnn_step_backward` 求出的对隐藏层偏移量导数的误差，并以 8 位有效数字的科学计数法显示。

下面我们来看 `rnn_step_backward` 函数的具体实现，代码如下：

```
27 def rnn_step_backward(dnext_h, cache):
28     """
29     递归神经网络 (RNN) 单步反向传播。
30     参数:
31     - dnext_h: 输出层误差反向传播到隐藏层得到的误差
32     - cache: 前向传播函数中缓存的变量
33     返回值元组:
34     - dx: 对输入信号的导数，形状为(迷你批次大小N, 输入信号维度D)
35     - dprev_h: 对前一时刻隐藏层输出的导数
36     - dWx: 对输入层到隐藏层连接权值的导数，形状为(迷你批次大小N,
37       隐藏层神经元个数H)
38     - dWh: 对隐藏层到隐藏层的导数，形状为(隐藏层神经元数量H,
39       隐藏层神经元数量H)
40     - db: 对隐藏层偏移量的导数，形状为(隐藏层神经元数量H)
41     """
42     dx, dprev_h, dWx, dWh, db = None, None, None, None, None
43     (x, Wx, Wh, prev_h, next_h, b) = cache
44     dnext_h_raw = (1 - next_h*next_h)*dnext_h
45     dx = np.dot(dnext_h_raw, Wx.T)
46     dprev_h = np.dot(dnext_h_raw, Wh.T)
47     dWx = np.dot(prev_h.T, dnext_h_raw)
48     dWh = np.dot(x.T, dnext_h_raw)
49     db = np.sum(dnext_h_raw, axis=0)
50     return dx, dprev_h, dWx, dWh, db
51
52
```

第 43 行：从缓存参数中取得单步前向传播时缓存的参数：输入信号 x ，输入层到隐藏层的连接权值矩阵 Wx ，隐藏层到隐藏层的连接权值矩阵 Wh ，前一时刻隐藏层输出值 `prev_h`，本时刻隐藏层输出值 `next_h`，隐藏层偏移量 b 。

第44行：求输出层对隐藏层输入的导数，公式为：

$$\frac{\partial y}{\partial \mathbf{z}^{(2)}} = \frac{\partial y}{\partial \mathbf{z}^{(3)}} \frac{\partial \mathbf{z}^{(3)}}{\partial \mathbf{a}^{(2)}} \frac{\partial \mathbf{a}^{(2)}}{\partial \mathbf{z}^{(2)}} = dy \cdot W_{hy} \cdot (1 - \mathbf{a}^{(2)} \cdot \mathbf{a}^{(2)})$$

式中，上标(3)代表输出层，(2)代表隐藏层， \mathbf{z} 代表输入， \mathbf{a} 代表输出。

$dnext_h$ 为输出层误差传输到隐藏层时形成的误差。因为在隐藏层使用的是双曲正切为激活函数的神经元，而双曲正切函数的导数为：

$$\tanh(x) = 1 - (\tanh(x))^2$$

第45行：求输出层对输入信号的导数，为隐藏层误差反向传播到输入层时的误差，即 dx 为隐藏层的误差与输入到隐藏层的连接权值矩阵进行点积计算的结果。

第46行：求输出层输出对上一时刻隐藏层输出的导数，公式推导如下：

$$\frac{\partial y}{\partial \mathbf{a}^{(2)(t-1)}} = \frac{\partial y}{\partial \mathbf{z}^{(3)(t)}} \frac{\partial \mathbf{z}^{(3)(t)}}{\partial \mathbf{a}^{(2)(t)}} \frac{\partial \mathbf{a}^{(2)(t)}}{\partial \mathbf{z}^{(2)(t)}} \frac{\partial \mathbf{z}^{(2)(t)}}{\partial \mathbf{a}^{(2)(t-1)}} = dnext_{h_{raw}} \cdot W_{hh}$$

根据链式求导法则，隐藏层 t 时刻输出对隐藏层 $t-1$ 时刻输出的导数，正好是隐藏层到隐藏层的连接矩阵。

第47行：求输出层输出对隐藏层到隐藏层连接权值导数，公式推导如下：

$$\begin{aligned} \frac{\partial y}{\partial W^{(2,2)}} &= \frac{\partial y}{\partial \mathbf{z}^{(3)(t)}} \frac{\partial \mathbf{z}^{(3)(t)}}{\partial \mathbf{a}^{(2)(t)}} \frac{\partial \mathbf{a}^{(2)(t)}}{\partial \mathbf{z}^{(2)(t)}} \frac{\partial \mathbf{z}^{(2)(t)}}{\partial W^{(2,2)}} = dnext_{h_{raw}} \cdot prev_h \\ &= dnext_{h_{raw}} \cdot hs[t-1] \end{aligned}$$

第48行：求输出层输出对输入层到隐藏层连接权值导数，公式推导如下：

$$\frac{\partial y}{\partial W^{(1,2)}} = \frac{\partial y}{\partial \mathbf{z}^{(3)(t)}} \frac{\partial \mathbf{z}^{(3)(t)}}{\partial \mathbf{a}^{(2)(t)}} \frac{\partial \mathbf{a}^{(2)(t)}}{\partial \mathbf{z}^{(2)(t)}} \frac{\partial \mathbf{z}^{(2)(t)}}{\partial W^{(1,2)}} = dnext_{h_{raw}} \cdot x$$

第49行：求隐藏层偏移量的导数。

测试程序入口如下：

```
342 if __name__ == '__main__':
343     print('RNN Image Caption Application')
344     test_rnn_step_backward()
```

运行结果为：

```
RNN Image Caption Application
dx error:7.40872242e-10
dprev_h error:5.21508571e-09
dWx error:7.05627275e-10
dWh error:1.38232291e-07
db error:3.40099497e-11
```

由上面的测试结果可以看出，`rnn_step_backward` 函数计算出来的参数的导数与利用数值计算出来的导数相差很小，因此可以认为我们对 `rnn_step_backward` 函数的实现是正确的。

6.2.5 完整前向传播

下面来看递归神经网络的整个正向传播过程，我们还是先写这个功能的测试用例，代码如下：

```

85 def test_rnn_forward():
86     N, T, D, H = 2, 3, 4, 5
87     x = np.linspace(-0.1, 0.3, num=N*T*D).reshape(N, T, D)
88     h0 = np.linspace(-0.3, 0.1, num=N*H).reshape(N, H)
89     Wx = np.linspace(-0.2, 0.4, num=D*H).reshape(D, H)
90     Wh = np.linspace(-0.4, 0.1, num=H*H).reshape(H, H)
91     b = np.linspace(-0.7, 0.1, num=H)
92
93     h, _ = rnn_forward(x, h0, Wx, Wh, b)
94     expected_h = np.asarray([
95         [
96             [-0.42070749, -0.27279261, -0.11074945, 0.05740409, 0.22236251],
97             [-0.39525808, -0.22554661, -0.0409454, 0.14649412, 0.32397316],
98             [-0.42305111, -0.24223728, -0.04287027, 0.15997045, 0.35014525],
99         ],
100         [
101             [-0.55857474, -0.39065825, -0.19198182, 0.02378408, 0.23735671],
102             [-0.27150199, -0.07088804, 0.13562939, 0.33099728, 0.50158768],
103             [-0.51014825, -0.30524429, -0.06755202, 0.17806392, 0.40333043],
104         ])
105     print('h error:%s' % format(rel_error(expected_h, h), '.8e'))
106

```

第 86 行：定义迷你批次大小 N 、序列长度 T 、输入信号维度 D 、隐藏层神经元数 H 。

第 87 行：在 $-0.1 \sim 0.3$ 之间等距取迷你批次 $N \times$ 序列长度 $T \times$ 输入信号维度 D 个数，并组成形状为（迷你批次 N ，序列长度 T ，输入信号维度 D ）的三维数组。

第 88 行： $t=0$ 时刻隐藏层输出值，在 $-0.3 \sim 0.1$ 之间等距离取迷你批次大小 $N \times$ 隐藏层神经元数量 H 个数字，并组成形状为（迷你批次大小 N ，隐藏层神经元数量 H ）的二维数组。

第 89 行：初始化输入层到隐藏层的连接权值矩阵。在 $-0.2 \sim 0.4$ 之间等距离取输入信号维度 $D \times$ 隐藏层神经元数量 H 个数，并组成形状为（输入信号维度 D ，隐藏层神经元数量 H ）的数组。

第 90 行：初始化隐藏层到隐藏层的连接权值矩阵。在 $-0.4 \sim 0.1$ 之间等距离取隐藏层神经元数量 $H \times$ 隐藏层神经元数量 H 个数，并组成形状为（隐藏层神经元数量 H ，隐藏层神经元数量 H ）的数组。

第 91 行：初始化隐藏层神经元偏移量：在 $-0.7 \sim 0.1$ 之间取隐藏层神经元数量 H 个数，组成一维数组。

第 93 行：调用 `rnn_forward` 函数，求出全部时刻隐藏层的输出值。

第 94～104 行：利用数值计算方法求出全部时刻隐藏层的输出值的正确值。

第 105 行：以科学计数法格式打印全部时刻隐藏层的计算输出值与正确输出值之间的误差。

在有了递归神经网络前向传播函数 `rnn_forward` 的测试用例之后，我们来看 `rnn_forward` 具体实现代码，如下：

```

53 def rnn_forward(x, h0, Wx, Wh, b):
54     """
55     完成所有训练样本集的前向传播过程，每个迷你训练样本集含有N个样本，每个样本
56     序列数T（共有T个时刻），每个序列为D维向量，隐藏层神经元数量为H。执行完
57     前向操作后，返回隐藏层所有样本所有时刻的输出值。
58     输入：
59     - x：整个迷你批次训练样本集，形状为（迷你批次大小N，序列数T，
60       输入向量维度D）
61     - h0：在t=0时刻隐藏层的初始状态，形状为（迷你批次大小N，
62       隐藏层神经元数量H）
63     - Wx：输入层到隐藏层的连接权值矩阵，形状为（迷你批次大小N，
64       隐藏层神经元数量H）
65     - Wh：隐藏层到隐藏层的连接权值矩阵，形状为（隐藏层神经元数量H，
66       隐藏层神经元数量H）
67     - b：隐藏层神经元偏移量向量，形状为（隐藏层神经元数量H）
68     返回值元组：
69     - h：所有样本所有时刻隐藏层的输出值张量，形状为（迷你批次大小N，
70       序列时刻数T，隐藏层神经元数量H）
71     - cache：为反向传播缓存的变量，包括：当前时刻输入样本x，输入层到隐藏层
72       连接权值矩阵Wx，隐藏层到隐藏层连接权值矩阵Wh，前一时刻隐藏层
73       输出值h0，当前时刻隐藏层输出值h，隐藏层神经元偏移量b
74     """
75     h, cache = None, None
76     N, T, D = x.shape
77     H = h0.shape[1]
78     h = np.zeros((N, T, H), dtype=h0.dtype)
79
80     cache = []
81     for t in range(T):
82         h[:, t, :], _ = rnn_step_forward(x[:, t, :], h0, Wx, Wh, b)
83         cache.append((x[:, t, :], Wx, Wh, h0, h[:, t, :], b))
84         h0 = h[:, t, :]
85     return h, cache
86
87

```

第78行：定义当前时刻隐藏层输出值张量 **h**，形状为（迷你批次大小 **N**，序列时刻数 **T**，隐藏层神经元数量 **H**），即一共有 **N** 个样本，每个样本有 **T** 个时刻，每个样本每个时刻隐藏层 **H** 个神经元的输出值。

第80行：定义缓存对象，为反向传播函数缓存对象，其为一列表，列表中每个元素为元组，为时刻 **t** 时的情况：输入信号 **x**、输入层到隐藏层连接权值矩阵 **Wx**、隐藏层到隐藏层连接权值 **Wh**、上一时刻隐藏层输出值、本时刻隐藏层输出值、隐藏层神经元偏移量。

第81行：对所有时刻进行循环。

第82行：调用 `rnn_step_forward` 函数，求出在当前时刻单步前向传播后，隐藏层输出值，这时输入值 `x[:, t, :]` 代表取 **t** 时刻后形成的二维数组（迷你批次大小 **N**，输入信号维度 **D**），这正好是 `rnn_step_forward` 函数的输入信号参数所需的格式。注：数组切片操作见第2章 Numpy 简易教程部分。

第83行：将下面的元组缓存到 `cache` 列表中，元组内容为时刻 **t** 时的情况：输入信号 **x**、输入层到隐藏层连接权值矩阵 **Wx**、隐藏层到隐藏层连接权值 **Wh**、上一时刻隐藏层输出值、本时刻隐藏层输出值、隐藏层神经元偏移量。

第84行：将本时刻新计算出来的隐藏层输出值赋给上一时刻隐藏层输出值，进行下一次循环体。

第85行：返回所有样本所有时刻隐藏层输出值和为反向传播函数缓存的变量。

完成 `rnn_forward` 函数之后，我们来看测试用例执行入口：

```

341 if __name__ == '__main__':
342     print('RNN Image Caption Application')
343     test_rnn_forward()

```


运行结果为：

```
RNN Image Caption Application
h error:7.72846616e-08
```

由上面的运行结果来看，计算的所有样本所有时刻隐藏层输出值与数值计算得到的正确值误差很小，这就说明我们对 `rnn_forward` 函数的实现是正确的。

6.2.6 完整反向传播

下面我们来看递归神经网络反向传播函数 `rnn_backward`，我们也先写测试用例，代码如下：

```
107 def test_rnn_backward():
108     N, D, T, H = 2, 3, 10, 5
109     x = np.random.randn(N, T, D)
110     h0 = np.random.randn(N, H)
111     Wx = np.random.randn(D, H)
112     Wh = np.random.randn(H, H)
113     b = np.random.randn(H)
114
115     out, cache = rnn_forward(x, h0, Wx, Wh, b)
116     dout = np.random.randn(*out.shape)
117     dx, dh0, dWx, dWh, db = rnn_backward(dout, cache)
118
119     fx = lambda x: rnn_forward(x, h0, Wx, Wh, b)[0]
120     fh0 = lambda h0: rnn_forward(x, h0, Wx, Wh, b)[0]
121     fWx = lambda Wx: rnn_forward(x, h0, Wx, Wh, b)[0]
122     fWh = lambda Wh: rnn_forward(x, h0, Wx, Wh, b)[0]
123     fb = lambda b: rnn_forward(x, h0, Wx, Wh, b)[0]
124
125     dx_num = eval_numerical_gradient_array(fx, x, dout)
126     dh0_num = eval_numerical_gradient_array(fh0, h0, dout)
127     dWx_num = eval_numerical_gradient_array(fWx, Wx, dout)
128     dWh_num = eval_numerical_gradient_array(fWh, Wh, dout)
129     db_num = eval_numerical_gradient_array(fb, b, dout)
130
131     print('dx error:%s' % format(rel_error(dx_num, dx), '.8e'))
132     print('dh0 error:%s' % format(rel_error(dh0_num, dh0), '.8e'))
133     print('dWx error:%s' % format(rel_error(dWx_num, dWx), '.8e'))
134     print('dWh error:%s' % format(rel_error(dWh_num, dWh), '.8e'))
135     print('db error:%s' % format(rel_error(db_num, db), '.8e'))
136
```

第 108 行：定义变量：迷你批次大小 N （包含样本数）、输入信号维度 D 、时刻数量 T 、隐藏层神经元数量 H 。

第 109 行：利用随机数初始化输入信号 x ，形状为（迷你批次大小 N ，时刻数量 T ，输入信号维度 D ）。

第 110 行：利用随机数初始化前一时刻隐藏层输出值，形状为（迷你批次大小 N ，隐藏层神经元数量 H ）。

第 111 行：利用随机数初始化输入层到隐藏层连接权值矩阵 W_x ，形状为（输入信号维度 D ，隐藏层神经元数量 H ）。

第 112 行：利用随机数初始化隐藏层到隐藏层连接权值矩阵 W_h ，形状为（隐藏层神经元数量 H ，隐藏层神经元数量 H ）。

第 113 行：利用随机数初始化隐藏层神经元偏移量 b ，形状为（隐藏层神经元数量 H ）。

第 115 行：调用 `rnn_forward` 函数求出所有样本所有时刻隐藏层输出值。

第 116 行：利用随机数生成输出层和上一时刻隐藏层误差反向传播到当前时刻隐藏层的误差值 dout。

第 117 行：调用 `rnn_backward` 计算出：对输入信号的导数 dx 、对前一时刻隐藏层输出的导数 $dh0$ 、对输入层到隐藏层连接权值的导数 dWx 、对隐藏层到隐藏层连接权值的导数 dWh 、对隐藏层神经元偏移量的导数 db 。

第 119~129 行：通过数值计算方法，求出：对输入信号的导数计算值 dx_num 、对前一时刻隐藏层输出的导数计算值 $dh0_num$ 、对输入层到隐藏层连接权值的导数计算值 dWx_num 、隐藏层到隐藏层连接权值的导数计算值 dWh_num 、对隐藏层神经元偏移量的导数计算值 db_num 。

第 131 行：采用科学计数法格式，打印 `rnn_backward` 函数返回的对输入信号的导数与正确值间的误差。

第 132 行：采用科学计数法格式，打印 `rnn_backward` 函数返回的对上一时刻隐藏层输出的导数与正确值间的误差。

第 133 行：采用科学计数法格式，打印 `rnn_backward` 函数返回的对输入层到隐藏层连接权值的导数与正确值间的误差。

第 134 行：采用科学计数法格式，打印 `rnn_backward` 函数返回的对隐藏层到隐藏层连接权值的导数与正确值间的误差。

第 135 行：采用科学计数法格式，打印 `rnn_backward` 函数返回的对隐藏层神经元偏移量的导数与正确值间的误差。

接下来就可以实现 `rnn_backward` 函数，代码如下：

```

88 def rnn_backward(dh, cache):
89     """
90     计算整个迷你批次全部时刻的反向传播过程。
91     参数：
92     - dh：输出层误差和上一时刻隐藏层误差反向传播到当前时刻隐藏层的误差，
93       形状为（迷你批次大小N，时刻数量T，隐藏层神经元数量H）
94     返回值：
95     - dx：对输入信号的导数，形状为（迷你批次中样本数N，时刻数量T，输入信号维度D）
96     - dh0：对上一时刻隐藏层输出值的导数，形状为（迷你批次中样本数N，
97       隐藏层神经元数量H）
98     - dWx：对输入层到隐藏层连接权值的导数，形状为（输入信号维度D，
99       隐藏层神经元数量H）
100    - dWh：对隐藏层到隐藏层连接权值的导数，形状为（隐藏层神经元数量H，
101      隐藏层神经元数量H）
102    - db：对隐藏层神经元偏移量的导数，形状为（隐藏层神经元数量H）
103    """
104    dx, dh0, dWx, dWh, db = None, None, None, None, None
105    N, T, H = dh.shape
106    (x, Wx, Wh, prev_h, next_h, b) = cache[0]
107    dx = np.zeros_like(x).repeat(T).reshape(x.shape[0], T, x.shape[1])
108    dh0 = np.zeros_like(prev_h)
109    dWx = np.zeros_like(Wx)
110    dWh = np.zeros_like(Wh)
111    db = np.zeros_like(b)
112    for t in reversed(range(T)):
113        dx[:, t, :], dprev_h, dWx_delta, dWh_delta, db_delta = \
114            rnn_step_backward(dh[:, t, :] + dh0, cache[t])
115        dWx += dWx_delta
116        dWh += dWh_delta
117        db += db_delta
118        dh0 = dprev_h
119    return dx, dh0, dWx, dWh, db
120
121

```

第 105 行：求出迷你批次中样本数 N ，时刻数量 T ，隐藏层神经元数量 H 。

第 106 行：从 `cache` 中取 $t=0$ 时刻变量：输入信号 x 、输入层到隐藏层连接权值 W_x 、隐藏层到隐藏层连接权值 W_h 、上一时刻隐藏层输出值 `prev_h`、当前时刻隐藏层输出值 `next_h`、隐藏层偏移量 b 。

第 107 行：定义全 0 张量 dx ，保存对输入信号的导数，形状为（迷你批次包含样本数 N ，时刻数量 T ，输入信号维度 D ）。

第 108 行：定义全 0 张量 dh_0 ，保存对上一时刻隐藏层输出值的导数，形状与上一时刻隐藏层输出值 `prev_h` 相同。

第 109 行：定义全 0 矩阵 dW_x ，保存对输入层到隐藏层连接权值的导数，形状与输入层到隐藏层连接权值矩阵 W_x 相同。

第 110 行：定义全 0 矩阵 dW_h ，保存对隐藏层到隐藏层连接权值的导数，形状与隐藏层到隐藏层连接权值矩阵 W_h 相同。

第 111 行：定义全 0 向量 db ，保存对隐藏层神经元偏移量的导数，形状与隐藏层偏移量 b 相同。

第 112 行：从 $t=T$ 时刻开始向前循环。

第 113、114 行：调用单步（某一时刻）反向传播函数 `rnn_step_backward`，求出对输入信号的导数、对前一时刻隐藏层输出值的导数 `dprev_h`、对输入层到隐藏层连接权值的导数 `dWx_delta`、对隐藏层到隐藏层连接权值的导数 `dWh_delta`、对隐藏层神经元偏移量的导数 `db_delta`。

第 115 行：将求出的对输入层到隐藏层连接权值的导数 `dWx_delta` 叠加到对输入层到隐藏层连接权值的导数 dW_x 上。

第 116 行：将求出的对隐藏层到隐藏层连接权值的导数 `dWh_delta` 叠加到对隐藏层到隐藏层连接权值的导数 dW_h 上。

第 117 行：将求出的对隐藏层神经元偏移量的导数 `db_delta` 叠加到对隐藏层神经元偏移量 db 上。

第 118 行：将计算出的对上一时刻隐藏层输出值的导数 `dprev_h` 赋给 dh_0 。

第 119 行：返回参数导数的计算结果。

下面我们来看测试用例运行程序入口：

```
341 if __name__ == '__main__':
342     print('RNN Image Caption Application')
343     test_rnn_backward()
```

运行结果为：

```
RNN Image Caption Application
dx error:1.32511775e-09
dh0 error:5.28867182e-10
dWx error:1.86103495e-08
dWh error:7.50301767e-08
db error:5.94189922e-10
```

由上面的测试结果可以看出，`rnn_backward` 函数返回的结果和用数值计算得出的正确结果之间的误差很小，可以证明我们的实现是正确的。

6.2.7 单词嵌入前向传播

在看具体程序之前，先简单介绍一下单词嵌入（Word Embedding）。

假如我们在做一个自然语言处理（NLP）应用，共有 10 个单词，并对其进行了编号：0——电脑；1——汽车；2——蛋白质；3——硬盘；4——火车；5——肽键；6——内存；7——轮船；8——酶；9——飞机。我们应该如何来表示这些词呢？

最简单的方式是所谓的 one-hot 方式，就是用一个 10 维向量来表示这些词，是哪个词就把哪个词对应位置的元素置为 1。例如我们想表示“火车”这个词，可以用向量 $[0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0]^T$ 来表示。这种表达方式的优点是非常简单，单词和索引具有一一对应关系，在前面介绍的递归神经网络简单应用中，就是采用这种方式表示的。但是这种方式也有很大的缺点，首先语言中常用的词至少有几万个，这样表示的话，矩阵将非常稀疏，浪费存储空间。其次，这些单词实际上并不是完全独立的，例如电脑与硬盘的关系，比电脑与汽车的关系要密切，而这种表示方式根本反映不出这种关系。

正是由于 one-hot 表示法有上述缺点，所以在很多系统中采用的是我们要介绍的单词嵌入方法，又称为词向量（word2vec）。其核心思想就是用低维向量来表示单词，单词之间的距离可以衡量单词的相似性。还以上段中的例子为例，假设我们以三维向量来表示这些词，例如硬盘、蛋白质、火车、酶这四个词的向量表示分别为 $[1.2, 2.1, 3.1]$ 、 $[2.1, 3.1, 4.1]$ 、 $[3.1, 4.5, 5.5]$ 、 $[2.1, 3.2, 4.2]$ 。这时硬盘与蛋白质的距离为 $= (2.1 - 1.2)^2 + (3.1 - 2.1)^2 + (4.1 - 3.1)^2 = 2.81$ ，而蛋白质与酶的距离为 $= (2.1 - 2.1)^2 + (3.2 - 3.1)^2 + (4.2 - 4.1)^2 = 0.02$ 。由此可以看到，由于硬盘和蛋白质几乎没什么关系，所以其距离就比较大，而酶本身就是蛋白质的一种，所以二者间的距离很小。

以上是随意给出的单词嵌入向量表示方式，如果有成千上万个单词，就需要计算机自动得到词向量的表示方式了，本节就来讨论这一问题的程序实现过程。

单词嵌入前向传播函数就是要求出给定单词序列（用 one-hot 方式表示的单词的索引值），以及这组单词的词向量表示方式。

同样，先写出 word_embedding_forward 函数的测试用例，代码如下：

```
137 def test_word_embedding_forward():
138     N, T, V, D = 2, 4, 5, 3
139     x = np.asarray([[0, 3, 1, 2], [2, 1, 0, 3]])
140     W = np.linspace(0, 1, num=V*D).reshape(V, D)
141
142     out, _ = word_embedding_forward(x, W)
143     expected_out = np.asarray([
144         [0., 0.07142857, 0.14285714],
145         [0.64285714, 0.71428571, 0.78571429],
146         [0.21428571, 0.28571429, 0.35714286],
147         [0.42857143, 0.5, 0.57142857]],
148         [[0.42857143, 0.5, 0.57142857],
149         [0.21428571, 0.28571429, 0.35714286],
150         [0., 0.07142857, 0.14285714],
151         [0.64285714, 0.71428571, 0.78571429]])
152
153     print('out error: %s' % format(rel_error(expected_out, out), '.e8'))
154
```

第 138 行：定义迷你批次中样本数 N、单词序列长度 T、单词数量 V、词向量维度 D。

第 139 行：定义输入信号 x ，形状为（迷你批次中样本数 N ，单词序列长度 T ），其值为每个单词的索引号。

第 140 行：定义参数矩阵 W ，从 0~1 之间等距离取单词数量 $V \times$ 词向量维度 D 个数，并形成二维数组，形状为（单词数量 V ，词向量维度 D ）。

第 142 行：调用 `word_embedding_forward` 函数，求出该单词序列所对应的词向量的表示形式。

第 143~151 行：定义正确的词向量表示形式。

第 153 行：采用科学计数法格式打印 `word_embedding_forward` 函数计算值与正确值之间的误差。

有了测试用例之后，我们来看 `word_embedding_forward` 函数的具体实现代码：

```
122 def word_embedding_forward(x, W):
123     """
124     单词嵌入前向传播函数，迷你批次中样本数N，每个样本中单词数为T，单词表中
125     单词数为V，词向量维数为D。
126     - x：要转换成词向量的单词序列，形状为（迷你样本中样本数N，每个样本中单
127       词数T），值为单词的索引号（在0到单词表中单词数V之间）。
128     - W：参数矩阵，形状为（单词表中单词数V，词向量维数D），给出了每个单词的
129       词向量，第一维是单词索引号，第二维为该单词的词向量
130     返回值元组：
131     - out：把输入信号由单词索引号表示的单词序列转换为用词向量表示的单词序列，
132       形状为（迷你批次中样本数N，每个样本中单词数T，词向量维数D）
133     - cache：缓存输入单词序列和单词表中单词数V
134     """
135     out, cache = None, None
136     N, T = x.shape
137     V, D = W.shape
138     out = np.zeros((N, T, D))
139     cache = (x, V)
140     out = W[x,:]
141     return out, cache
142
143
```

第 136 行：取出迷你批次中样本数 N 、每个样本中单词数 T 。

第 137 行：取出单词表中单词数 V 、词向量维度 D 。

第 138 行：初始化存储以词向量表示的单词序列变量。

第 139 行：将以单词索引号表示的单词序列 x 和单词索引号对应的词向量矩阵 W 保存到 `cache` 中，供单词嵌入反向传播函数使用。

第 140 行：根据单词序列中的单词索引号求出对应的词向量，并保存到 `out` 变量中。

第 141 行：返回以词向量形式表示的单词序列和缓存给单词嵌入反向传播函数的参数。

测试用例运行入口如下：

```
341 if __name__ == '__main__':
342     print('RNN Image Caption Application')
343     test_word_embedding_forward()
```

运行结果为：

```
RNN Image Caption Application
out error: 1.00000001e-08
```

6.2.8 单词嵌入反向传播

我们先来写单词嵌入反向传播 word_embedding_backward 函数的测试用例，代码如下：

```
155 def test_word_embedding_backward():
156     N, T, V, D = 50, 3, 5, 6
157     x = np.random.randint(V, size=(N, T))
158     W = np.random.randn(V, D)
159     out, cache = word_embedding_forward(x, W)
160     dout = np.random.randn(*out.shape)
161     dW = word_embedding_backward(dout, cache)
162     f = lambda W: word_embedding_forward(x, W)[0]
163     dW_num = eval_numerical_gradient_array(f, W, dout)
164     print('dW error: %s' % format(rel_error(dW, dW_num), '.8e'))
165
166
```

第 156 行：定义迷你批次中样本数 N、每个样本中单词数 T、单词表中单词数 V、词向量维度 D。

第 157 行：定义输入值 x，以单词索引号表示的单词序列，形状为（迷你批次中样本数 N，每样本中单词数 T），元素值为单词索引号。

第 158 行：定义词向量矩阵 W，保存每个单词索引号对应的词向量。

第 159 行：调用 word_embedding_forward，生成以词向量表示的单词序列。

第 160 行：利用随机数生成词向量嵌入层误差。

第 161 行：求出单词索引号对应词向量矩阵 W 的导数。

第 162、163 行：利用数值方法求出单词索引号对应词向量矩阵 W 的导数的计算值。

第 164 行：以科学计数法格式打印 word_embedding_backward 函数的返回值与数值计算得到的正确值之间的误差。

下面我们来看 word_embedding_backward 函数的具体实现，在阅读具体代码之前，先举一个具体的例子，看看这个函数的具体工作。

假设输入序列如表 6.1 所示。

表 6.1 单词序列

	0	1	2	3
0	0	3	1	2
1	2	1	0	3

误差为 dout，如表 6.2 所示。

表 6.2 词向量误差

	0	1	2	3
0	[0.1, 0.1, 0.1]	[0.2, 0.2, 0.2]	[0.3, 0.3, 0.3]	[0.4, 0.4, 0.4]
1	[0.01, 0.01, 0.01]	[0.02, 0.02, 0.02]	[0.03, 0.03, 0.03]	[0.04, 0.04, 0.04]

表中行的方向为迷你批次中的样本序号，列的方向为每个样本中的单词序号。

该函数返回值格式如表 6.3 所示。

表 6.3 word_embedding_backward函数返回值

单词序号	误 差	备 注
0	[0.13, 0.13, 0.13]	单词索引号为0的词出现在(0, 0)和(1, 2)中，将dout表中对应位置的值相加：0.1+0.03=0.13
1	[0.32, 0.32, 0.32]	单词索引号为1的词出现在(0, 2)和(1, 1)中，将dout表中对应位置的值相加：0.3+0.02=0.32
2	[0.41, 0.41, 0.41]	单词索引号为2的词出现在(1, 0)和(0, 3)中，将dout表中对应位置的值相加：0.01+0.4=0.41
3	[0.24, 0.24, 0.24]	单词索引号为3的词出现在(0, 1)和(1, 3)中，将dout表中对应位置的值相加：0.2+0.04=0.24
4	[0.0, 0.0, 0.0]	本单词未在单词序列中出现

下面我们来看具体程序实现，代码如下：

```
144 def word_embedding_backward(dout, cache):
145     """
146     单词嵌入反向传播过程，将给定单词序列对应词向量的误差，按单词索引号进行
147     累加并返回,详细情况见书中说明。
148     参数：
149     - dout：上层传过来的误差，单词序列中每个词在词向量每个维度上的误差，形状
150       为（迷你批次中样本数N，每个样本中单词数T，词向量维度D）
151     - cache：缓存数据，原始单词序列和单词表中单词数
152     返回值：
153     - dW：每个词向量对这个单词序列而言的误差之和
154     """
155     dW = None
156     N, T, D = dout.shape
157     x, V = cache
158     dW = np.zeros((V, D))
159     np.add.at(dW, x, dout)
160     return dW
161
162
```

第 156 行：求出迷你批次中样本数 N、每个样本中单词数 T、词向量维度 D。

第 157 行：从 cache 中取出原始的单词序列和单词表中单词数 V。

第 158 行：定义词向量导数变量 dW，在给定的单词序列下每个词向量的累积误差。

第 159 行：单词序列 x 的每个元素对应一个单词索引号，由此可以在 dW 中找到对应的词向量，按照 x 的下标位置找到 dout 中词向量误差，将这个误差累加到 dW 对应的词向量误差中。

测试程序入口为：

```
341 if __name__ == '__main__':
342     print('RNN Image Caption Application')
343     test_word_embedding_backward()
```

运行结果为：

```
RNN Image Caption Application
dW error: 3.27634862e-12
```

由上面的输出可以看到，word_embedding_backward 函数返回值与数值计算的正确值之间的误差非常小，所以我们的函数实现是正确的。

6.2.9 输出层前向/反向传播

下面我们来开发从隐藏层到输出层的正向传播，先开发测试用例，代码如下：

```

167 def test_temporal_affine_forward_backward():
168     # Gradient check for temporal affine layer
169     N, T, D, M = 2, 3, 4, 5
170
171     x = np.random.randn(N, T, D)
172     w = np.random.randn(D, M)
173     b = np.random.randn(M)
174
175     out, cache = temporal_affine_forward(x, w, b)
176
177     dout = np.random.randn(*out.shape)
178
179     fx = lambda x: temporal_affine_forward(x, w, b)[0]
180     fw = lambda w: temporal_affine_forward(x, w, b)[0]
181     fb = lambda b: temporal_affine_forward(x, w, b)[0]
182
183     dx_num = eval_numerical_gradient_array(fx, x, dout)
184     dw_num = eval_numerical_gradient_array(fw, w, dout)
185     db_num = eval_numerical_gradient_array(fb, b, dout)
186
187     dx, dw, db = temporal_affine_backward(dout, cache)
188
189     print('dx error: %s' % format(rel_error(dx_num, dx), '.8e'))
190     print('dw error: %s' % format(rel_error(dw_num, dw), '.8e'))
191     print('db error: %s' % format(rel_error(db_num, db), '.8e'))
192

```

第 169 行：定义迷你批次中样本数 N 、序列长度 T 、输入信号维度 D 、输出层神经元数 M 。

第 171 行：用随机数定义输入信号，形状为（迷你批次中样本数 N ，序列长度 T ，输入信号维度 D ）。

第 172 行：用随机数初始化隐藏层到输出层连接权值矩阵 w 。

第 173 行：用随机数初始化输出层神经元偏移量 b 。

第 175 行：调用 `temporal_affine_forward` 函数，求出输出层输出。

第 177 行：用随机数定义输出层误差。

第 179~185 行：用数值计算方法求出对输出层输入的导数计算值 dx_num 、对隐藏层到输出层连接权值的导数计算值 dw_num 、对输出层神经元偏移量的导数计算值 db_num 。

第 187 行：调用 `temporal_affine_backward` 函数，求出对输出层输入的导数 dx 、对隐藏层到输出层连接权值的导数 dw 、对输出层神经元偏移量的导数 db 。

第 189 行：采用科学计数法格式打印对输出层输入的导数 `temporal_affine_backward` 函数返回值与正确值间的误差。

第 190 行：采用科学计数法格式打印对隐藏层到输出层连接权值的导数 `temporal_affine_backward` 函数返回值与正确值间的误差。

第 191 行：采用科学计数法格式打印对输出层神经元偏移量的导数 `temporal_affine_backward` 函数返回值与正确值间的误差。

下面我们来看隐藏层到输出层的前向传播函数 `temporal_affine_forward` 的具体实现，代码如下：


```

305 def temporal_affine_forward(x, w, b):
306     """
307     隐藏层到输出层的正向传播，迷你批次中样本数为N，每个序列长度为T，输入信号
308     维度为D，经仿射变换将隐藏层输出转化为输出层输出，其形状为（迷你批次样本
309     数为N，每个序列长度为T，输出层神经元数为M）
310     参数：
311     - x：输出层的输入信号，形状为（迷你批次中样本数N，序列长度T，输入信号维度
312       即隐藏层神经元数D）
313     - w：隐藏层到输出层的连接权值，形状为（隐藏层神经元数D，输出层神经元数M）
314     - b：输出层神经元偏移量，形状为（输出层神经元数M）
315     返回值元组：
316     - out：输出层输出，形状为（迷你批次中样本数N，序列长度T，输出层神经元数M）
317     - cache：缓存输出层输入，隐藏层到输出层连接权值矩阵w，输出层神经元偏移量b，
318       输出层输出out
319     """
320     N, T, D = x.shape
321     M = b.shape[0]
322     out = x.reshape(N * T, D).dot(w).reshape(N, T, M) + b
323     cache = x, w, b, out
324     return out, cache
325
326

```

第 320 行：求出迷你批次中样本数 N、序列长度 T、输出层输入信号维度（隐藏层神经元数）D。

第 321 行：求出输出层神经元数量 M。

第 322 行：用仿射变换求出输出层输出。

第 323 行：缓存输出层输入信号 x、隐藏层到输出层连接权值矩阵 w、输出层神经元偏移量 b、输出层输出 out。

第 324 行：返回输出层输出和缓存。

下面来看输出层到隐藏层的反向传播 temporal_affine_backward 函数实现，代码如下：

```

327 def temporal_affine_backward(dout, cache):
328     """
329     输出层误差向隐藏层反向传播。
330     输入：
331     - dout：输出层误差，形状为（迷你批次中样本数N，序列长度T，输出层神经元数M）
332     - cache：接收前向传播阶段缓存的变量
333     返回值元组：
334     - dx：对输出层输入的导数，形状为（迷你批次中样本数N，序列长度T，输出层
335       输入信号维度即隐藏层神经元数D）
336     - dw：对隐藏层到输出层连接权值的导数，形状为（输出层输入信号维度即隐藏层
337       神经元数D，输出层神经元数M）
338     - db：对输出层神经元偏移量的导数，形状为（输出层神经元数M）
339     """
340     x, w, b, out = cache
341     N, T, D = x.shape
342     M = b.shape[0]
343
344     dx = dout.reshape(N * T, M).dot(w.T).reshape(N, T, D)
345     dw = dout.reshape(N * T, M).T.dot(x.reshape(N * T, D)).T
346     db = dout.sum(axis=(0, 1))
347
348     return dx, dw, db
349
350

```

第 340 行：从 cache 参数中取出输出层输入信号 x、隐藏层到输出层连接权值矩阵 w、输出层神经元偏移量 b、输出层输出 out。

第 341 行：求出迷你批次中样本数 N、序列长度 T、输出层输入信号维度（隐藏层神经元数）D。

第 342 行：求出输出层神经元数 M。

第 344 行：求对输出层输入的导数 dx。

第 345 行：求对隐藏层到输出层连接权值的导数 dw 。

第 346 行：求对输出层神经元偏移量的导数 db 。

第 348 行：返回对输出层输入的导数 dx 、对隐藏层到输出层连接权值的导数 dw 、对输出层神经元偏移量的导数 db 。

测试程序入口为：

```
341 if __name__ == '__main__':
342     print('RNN Image Caption Application')
343     test_temporal_affine_forward_backward()
```

运行结果为：

```
RNN Image Caption Application
dx error: 1.73275728e-11
dw error: 4.59562628e-11
db error: 8.43936220e-12
```

6.2.10 输出层代价函数计算

我们先写出输出层代价函数值计算的测试用例，代码如下：

```
193 def test_temporal_softmax_loss():
194     N, T, V = 100, 1, 10
195
196     def check_loss(N, T, V, p):
197         x = 0.001 * np.random.randn(N, T, V)
198         y = np.random.randint(V, size=(N, T))
199         mask = np.random.rand(N, T) <= p
200         print('temporal_softmax_loss: %f' % \
201               temporal_softmax_loss(x, y, mask)[0])
202
203     check_loss(100, 1, 10, 1.0) # Should be about 2.3
204     check_loss(100, 10, 10, 1.0) # Should be about 23
205     check_loss(5000, 10, 10, 0.1) # Should be about 2.3
206
```

第 194 行：定义迷你批次中样本数 N 、序列长度 T 、单词表中单词数 V 。

第 196 行：定义代价函数值检验函数。

❑ N ：迷你批次中样本数。

❑ T ：序列长度。

❑ V ：单词表中单词数。

❑ p ：概率阈值。

第 197 行：定义输入信号（输出层输出） x ，第 i 个样本第 j 个位置为对所有单词的分数（出现概率）。

第 198 行：定义目标输出值，第 i 个样本第 j 个位置出现单词的索引号。

第 199 行：对第 i 个样本第 j 个位置产生的 $0\sim 1$ 之间的随机数小于等于参数概率时该元素为真，否则为假。

第 200、201 行：调用 `temporal_softmax_loss` 函数计算代价函数值，并用科学计数法显示该值。

第 203~205 行：调用 `check_loss` 函数 3 次，检查是否打印出接近最右侧的数值。

下面我们来看 `temporal_softmax_loss` 函数具体实现，代码如下：

```

351 def temporal_softmax_loss(x, y, mask, verbose=False):
352     """
353     递归神经网络（RNN）的softmax层，输入信号为RNN输出层输出信号，格式为第i个
354     样本第j个位置为一个v维向量，元素值为每个单词RNN预测的出现概率。y为目标
355     输出结果，格式为第i个样本第j个位置为单词索引号。我们采用交叉熵代价函数，
356     将所有代价函数值进行累加，最后再在迷你批次里进行平均。
357     mask表示是否忽略第i个样本第j个位置元素，如果忽略则在最终的序列中将显示
358     为NULL。因此我们可以通过mask指定哪些元素需要包含在代价函数计算中。
359     参数：
360     - x：输入信号，即为RNN输出层的输出信号，格式为第i个样本第j个位置为一个
361       v维向量，元素值为每个单词RNN预测的出现概率。形状为（迷你批次中样本
362       数N，序列长度T，单词表中单词数V）
363     - y：目标输出值，格式为第i个样本第j个位置应该出现单词的索引号。形状为
364       （迷你批次中样本数N，序列长度T）
365     - mask：元素是否需要包含在代价函数计算过程中，格式为第i个样本第j个位置
366       代表该元素是否应该包含在代价函数计算中。形状为（迷你批次中样本
367       数N，序列长度T）
368     返回值元组：
369     - loss：代价函数值
370     - dx：代价函数对输入的导数
371     """
372
373     N, T, V = x.shape
374
375     x_flat = x.reshape(N * T, V)
376     y_flat = y.reshape(N * T)
377     mask_flat = mask.reshape(N * T)
378
379     probs = np.exp(x_flat - np.max(x_flat, axis=1, keepdims=True))
380     probs /= np.sum(probs, axis=1, keepdims=True)
381     loss = -np.sum(mask_flat * np.log(probs[np.arange(N * T), y_flat])) / N
382     dx_flat = probs.copy()
383     dx_flat[np.arange(N * T), y_flat] -= 1
384     dx_flat /= N
385     dx_flat *= mask_flat[:, None]
386
387     if verbose: print('dx_flat: %s' % dx_flat.shape)
388
389     dx = dx_flat.reshape(N, T, V)
390
391     return loss, dx

```

第 373 行：求出迷你批次中样本数 N 、序列长度 T 、单词表中单词数量 V 。

第 375 行：将输入信号 x 变为二维数组，第 1 维为迷你批次中的样本数 $N \times$ 序列长度 T ，第 2 维是单词表中单词数 V 维的向量。

第 376 行：将目标信号变为一维数组，长度为迷你批次中的样本数 $N \times$ 序列长度 T ，元素值为单词的索引号。

第 377 行：将 `mask` 变为一维数组，长度为迷你批次中的样本数 $N \times$ 序列长度 T ，元素为每个位置是否包括在代价函数计算中。

第 379 行：求出概率列表。二维数组 `x_flat` 每一列为一个单词，元素第 i 个样本第 j 个位置为该单词出现的概率，先将每个元素值减去最大的概率值，然后取指数值。

第 380 行：将概率数组每项除以各项之和。

第 379 和 380 行用公式表示为：

$$p_i = \frac{e^{p_i - p_{\max}}}{\sum_{j=1}^{N \times T} e^{p_j - p_{\max}}}$$

第 381 行：对第 i 个样本第 j 个位置，先通过 `y_flat` 找到是哪个单词的索引号，从而得到其出现概率，然后取对数并加负号，结合 `mask_flat` 决定是否需要包含在代价函数计算中，

将这些值叠加并除以样本数求平均值。

第 382 行：复制概率数组 `probs` 到输入信号导数的数组，并将其变为一维数组 `dx_flat`。

第 383 行：将第 i 个样本第 j 个位置正确单词位置-1，因为按照 `softmax` 规定，是正确单词时输出概率为 1，否则输出为 0，所以本行代码是求出 `softmax` 行的输出误差。

第 384 行：除以迷你批次中样本数 N 并取平均值。

第 385 行：与 `mask_flat` 相乘，去掉不包含在计算代价函数中的项。

第 389 行：改变形状为（迷你批次中样本数 N ，序列长度 N ，单词表中单词数 V ）。

第 391 行：返回代价函数值和对输入信号的导数值。

测试程序入口为：

```
341 if __name__ == '__main__':
342     print('RNN Image Caption Application')
343     test_temporal_softmax_loss()
```

运行结果为：

```
RNN Image Caption Application
temporal_softmax_loss: 2.302473
temporal_softmax_loss: 23.025985
temporal_softmax_loss: 2.317802
```

由上面的结果可以看出，打印值与希望值基本一致，所以证明求代价函数值部分的程序是正确的。

再写出输出层对输入信号的导数测试用例，代码如下：

```
207 def test_temporal_softmax_loss1():
208     # Gradient check for temporal softmax loss
209     N, T, V = 7, 8, 9
210
211     x = np.random.randn(N, T, V)
212     y = np.random.randint(V, size=(N, T))
213     mask = (np.random.rand(N, T) > 0.5)
214
215     loss, dx = temporal_softmax_loss(x, y, mask, verbose=False)
216
217     dx_num = eval_numerical_gradient(lambda x: temporal_softmax_loss(\
218                                     x, y, mask)[0], x, verbose=False)
219
220     print('dx error: %s' % format(rel_error(dx, dx_num), '.8e'))
221
```

第 209 行：定义迷你批次中样本数 N 、序列长度 T 、单词表中单词数 V 。

第 211 行：定义输入信号 x ，第 i 个样本第 j 个位置为单词表中单词数 V 向量，用随机数填充，表示 RNN 网络输出层输出。

第 212 行：定义期望输出结果，第 i 个样本第 j 个位置，值为 0 到单词表中单词数 V 的随机数，代表正确的单词的索引号。

第 213 行：定义 `mask`，第 i 个样本第 j 个位置值为看本次生成的随机数是否大于 0.5，如果大于则为真，否则为假。为真时对应第 i 个样本第 j 个位置包含在代价函数计算中。

第 215 行：调用 `temporal_softmax_loss` 函数，计算代价函数值和对输入信号的导数。

第 217 行：利用数值计算方法求出该导数的计算值。

第 220 行：采用科学计数法打印对 `softmax` 层输入信号导数、`temporal_softmax_loss` 函数返回值与正确值间的误差。

测试程序入口为:

```
342 if __name__ == '__main__':
343     print('RNN Image Caption Application')
344     test_temporal_softmax_loss1()
```

运行结果为:

```
RNN Image Caption Application
dx error: 2.96677914e-08
```

由打印结果可以看出，`temporal_softmax_loss` 函数的返回结果，与通过数值计算出的结果误差很小，这说明我们的实现是正确的。

6.2.11 图像标注网络整体架构

我们已经完成了用递归神经网络做图像标注的准备工作，下面就可以利用这些功能来搭建图像标注应用网络了。

图像标注的网络架构和工作流程如图 6.4 所示。

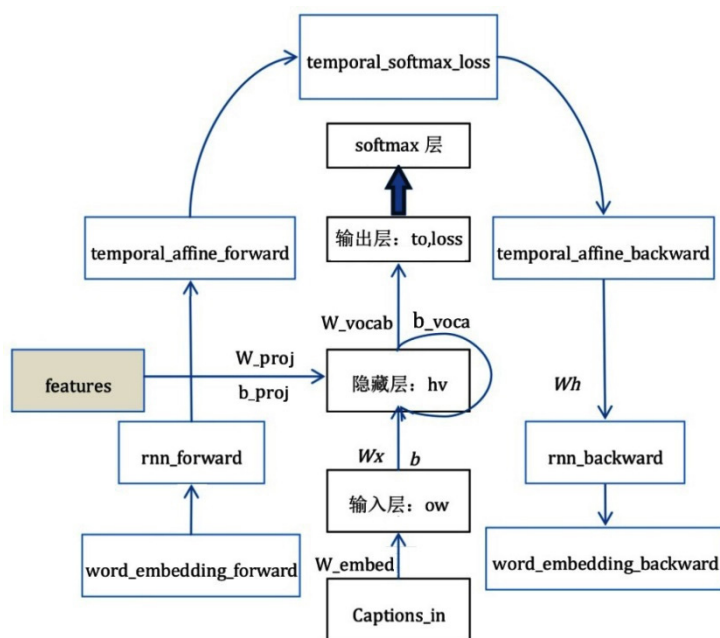


图 6.4 图像标注的网络架构和工作流程

图像标注工作流程如下:

- (1) 用一组单词序列作为输入。

(2) 以连接权值矩阵 $\mathbf{W}_{\text{embed}}$ 为参数，调用 `word_embedding_forward` 函数，生成值作为递归神经网络的输入值。

(3) 以输入层到隐藏层连接权值矩阵 W_x 和隐藏层神经元偏移量 b 为参数, 调用 `rnn forward` 方法, 求出隐藏层的输出。

(4) 通过卷积神经网络得到图像特征，以连接权值矩阵 W_{proj} 和偏移量向量 b_{proj} 为参数进行线性变换，作为隐藏层第 0 时刻的值。

(5) 以隐藏层到输出层连接权值矩阵 W_{vocab} 和输出层神经元偏移量 b_{vocab} 为参数，调用 `temporal_affine_forward` 函数，计算输出层输出。

(6) 输出层输出直接作为 softmax 层输入，调用 `temporal_softmax_loss` 函数，计算代价函数和相对于输入信号的导数。

(7) 调用 `temporal_affine_backward` 函数，将误差反向传播到隐藏层，并求各参数的导数。

(8) 调用 `rnn_backward` 函数，将误差反向传播到输入层，并求各相关参数的导数。

(9) 调用 `word_embedding_backward` 函数，将误差传播到 Captions 层，并求相关参数的导数。

6.2.12 代价函数计算

我们需要引入 CaptionRNN 类来表示上一节所描述的网络，CaptionRNN 类定义代码如下：

```
1 import numpy as np
2 from cs231n.layers import *
3 from cs231n.rnn_layers import *
4
5 class CaptioningRNN(object):
6     """
7     根据由卷积神经网络 (CNN) 提取出来的图像特征，通过递归神经网络给出
8     图像标注。
9     我们规定：图像特征维度为D维，单词表中单词数为V，标注序列长度为T，隐
10    藏层神经元数为H，词向量维数为W，迷你批次中样本数为N。
11    注：本网络实现中未使用任意调整项，如权值衰减等。
12    """
13
14    def __init__(self, word_to_idx, input_dim=512, wordvec_dim=128,
15                 hidden_dim=128, cell_type='rnn', dtype=np.float32):
16        """
17        生成CaptionRNN类实例。
18        输入：
19        - word_to_idx：单词到索引号字典
20        - input_dim：图像特征向量维度D
21        - wordvec_dim：词向量维度W
22        - hidden_dim：隐藏层神经元数H
23        - cell_type：网络类型，有普通RNN或长短期记忆LSTM两种
24        - dtype：训练中用np.float32，数值验证时用np.float64
25        """
26        if cell_type not in {'rnn', 'lstm'}:
27            raise ValueError('Invalid cell_type "%s"' % cell_type)
28
29        self.cell_type = cell_type
30        self.dtype = dtype
31        self.word_to_idx = word_to_idx
32        self.idx_to_word = {i: w for w, i in word_to_idx.iteritems()}
33        self.H = hidden_dim
34        self.params = {}
35
36        vocab_size = len(word_to_idx)
37
38        self._null = word_to_idx['<NULL>']
39        self._start = word_to_idx.get('<START>', None)
40        self._end = word_to_idx.get('<END>', None)
41
42        # Initialize word vectors
43        self.params['W_embed'] = np.random.randn(vocab_size, wordvec_dim)
```

```

44     self.params['W_embed'] /= 100
45
46     # Initialize CNN -> hidden state projection parameters
47     self.params['W_proj'] = np.random.randn(input_dim, hidden_dim)
48     self.params['W_proj'] /= np.sqrt(input_dim)
49     self.params['b_proj'] = np.zeros(hidden_dim)
50
51     # Initialize parameters for the RNN
52     dim_mul = {'lstm': 4, 'rnn': 1}[cell_type]
53     self.params['Wx'] = np.random.randn(wordvec_dim, dim_mul * hidden_dim)
54     self.params['Wx'] /= np.sqrt(wordvec_dim)
55     self.params['Wh'] = np.random.randn(hidden_dim, dim_mul * hidden_dim)
56     self.params['Wh'] /= np.sqrt(hidden_dim)
57     self.params['b'] = np.zeros(dim_mul * hidden_dim)
58
59     # Initialize output to vocab weights
60     self.params['W_vocab'] = np.random.randn(hidden_dim, vocab_size)
61     self.params['W_vocab'] /= np.sqrt(hidden_dim)
62     self.params['b_vocab'] = np.zeros(vocab_size)
63
64     # Cast parameters to correct dtype
65     for k, v in self.params.items():
66         self.params[k] = v.astype(self.dtype)
67
68

```

这个类的参数比较多，下面我们逐行进行讲解。

第 26、27 行：验证网络类型，为 RNN 或 LSTM 中的一种。

第 29 行：定义属性 `cell_type`，判断是 RNN 网络还是 LSTM 网络。

第 30 行：定义属性 `dtype`，训练时用 `np.float32`，数值验证时用 `np.float64`。

第 31 行：定义属性 `word_to_idx` 字典，通过单词查到索引号。

第 32 行：定义属性 `idx_to_word` 字典，通过单词索引号找到对应单词。

第 33 行：定义属性 `H`，为隐藏层神经元数量。

第 34 行：定义属性 `params` 字典，用于保存网络参数。

第 36 行：求出单词表中单词总数量 `vocab_size`。

第 38 行：定义空字符在标注序列中的格式。

第 39 行：定义属性 `_start`，表示标注序列中的开始符号。

第 40 行：定义属性 `_end`，表示标注序列中的结束符号。

第 43、44 行：向属性 `params` 字典中添加 `W_embed` 参数，其保存所有单词的词向量，用于为 RNN 输入层准备数据。

第 47~49 行：向属性 `params` 字典中添加 `W_proj` 和 `b_proj` 参数，用于将经卷积神经网络提取的图像特征仿射变换为隐藏层第 0 时刻的值。

第 52 行：定义 `dim_mul` 变量，因为当网络为 LSTM 时，连接权值矩阵维度为当网络是 RNN 时的 4 倍。

第 53、54 行：向属性 `params` 字典中加入 `Wx`，为输入层到隐藏层连接权值矩阵，用足够小的随机数进行初始化。

第 55、56 行：向属性 `params` 字典中加入 `Wh`，为隐藏层到隐藏层连接权值矩阵，用足够小的随机数进行初始化。

第 57 行：向属性 `params` 字典中加入 `b`，为隐藏层神经元偏移量，用 0 来初始化。

第 60 行：向属性 `params` 字典中加入 `W_vocab`，为隐藏层到输出层的连接权值矩阵，用足够小的随机数进行初始化。

第 61 行：向属性 `params` 字典中加入 `b_vocab`，为输出层神经元偏移量，用 0 来初始化。

第 65、66 行：设置所有参数的数据类型。

下面来写 `CaptionRNN` 模型代价函数值计算的测试用例，代码如下：

```
222 def test_rnn_loss():
223     N, D, W, H = 10, 20, 30, 40
224     word_to_idx = {'<NULL>': 0, 'cat': 2, 'dog': 3}
225     V = len(word_to_idx)
226     T = 13
227
228     model = CaptioningRNN(word_to_idx,
229                           input_dim=D,
230                           wordvec_dim=W,
231                           hidden_dim=H,
232                           cell_type='rnn',
233                           dtype=np.float64)
234
235     # Set all model parameters to fixed values
236     for k, v in model.params.iteritems():
237         model.params[k] = np.linspace(-1.4, 1.3, num=v.size).\
238             reshape(*v.shape)
239
240     features = np.linspace(-1.5, 0.3, num=(N * D)).reshape(N, D)
241     captions = (np.arange(N * T) % V).reshape(N, T)
242
243     loss, grads = model.loss(features, captions)
244     expected_loss = 9.83235591003
245
246     print('loss: %f' % loss)
247     print('expected loss: %f' % expected_loss)
248     print('difference: %s' % format(abs(loss - expected_loss), '.8e'))
249     print('grads:%f' % grads)
250
```

第 223 行：定义迷你批次中样本数 `N`、图像特征向量维度 `D`、词向量维度 `W`、隐藏层神经元数量 `H`。

第 224 行：定义单词到索引号的字典。

第 225 行：定义单词表中单词数 `V`。

第 226 行：图像标注长度 `T`。

第 228~233 行：初始化 `CaptioningRNN` 类对象，指定网络类型为 `rnn`，数值类型为 `np.float64`。

第 236~238 行：为所有参数赋固定值。

第 240 行：定义图像特征向量。

第 241 行：定义原始标注序列，在本例中为(0,0)=0, (0, 1)=1, (0, 2)=2, (0,3)=0, (0, 4)=1, 依次类推。

第 243 行：调用 `CaptioningRNN` 类的 `loss` 方法，求出代价函数值和对输入的导数。

第 244 行：定义代价函数值的正确值。

第 246 行：打印从 `CaptioningRNN` 类的 `loss` 方法返回的代价函数值。

第 247 行：打印正确的代价函数值。

第 248 行：打印采用科学计数法格式的代价函数值分别为 `CaptioningRNN` 类的 `loss` 方法返回值和正确值之间的误差。

第 249 行：打印 `CaptioningRNN` 类的 `loss` 方法返回的对输入信号的导数。

下面我们来看 `CaptioningRNN.loss` 方法的具体实现，代码如下：


```

69 def loss(self, features, captions):
70     """
71     计算模型的代价函数值和对所有参数的导数。输入为图像特征向量和正确的标注
72     序列，系统采用RNN或LSTM网络进行学习。
73     参数：
74     - features：图像特征向量，形状为（迷你批次中样本数N，图像特征向量维数D）
75     - captions：正确的标注序列，形状为（迷你批次中样本数N，标注序列长度T），
76       元素值为单词在单词表中的索引号。
77     返回值元组：
78     - loss：代价函数值，是一个标量
79     - grads：对所有参数的导数
80     """
81     # Cut captions into two pieces: captions_in has everything but the last word
82     # and will be input to the RNN; captions_out has everything but the first
83     # word and this is what we will expect the RNN to generate. These are offset
84     # by one relative to each other because the RNN should produce word (t+1)
85     # after receiving word t. The first element of captions_in will be the START
86     # token, and the first element of captions_out will be the first word.
87     captions_in = captions[:, :-1]
88     captions_out = captions[:, 1:]
89
90     # You'll need this
91     mask = (captions_out != self._null)
92
93     # Weight and bias for the affine transform from image features to initial
94     # hidden state
95     W_proj, b_proj = self.params['W_proj'], self.params['b_proj']
96
97     # Word embedding matrix
98     W_embed = self.params['W_embed']
99
100    # Input-to-hidden, hidden-to-hidden, and biases for the RNN
101    Wx, Wh, b = self.params['Wx'], self.params['Wh'], self.params['b']
102
103    # Weight and bias for the hidden-to-vocab transformation.
104    W_vocab, b_vocab = self.params['W_vocab'], self.params['b_vocab']
105
106    loss, grads = 0.0, {}
107    N, D = features.shape
108    T = captions.shape[1]
109    H = self.H
110    h0 = np.dot(features, W_proj) + b_proj
111    ow, ow_cache = word_embedding_forward(captions_in, W_embed)
112    hv, hv_cache = None, None
113    if 'rnn' == self.cell_type:
114        hv, hv_cache = rnn_forward(ow, h0, Wx, Wh, b)
115    else:
116        print('lstm')
117    to, to_cache = temporal_affine_forward(hv, W_vocab, b_vocab)
118    loss, dx = temporal_softmax_loss(to, captions_out, mask)
119
120    dhv, dW_vocab, db_vocab = temporal_affine_backward(dx, to_cache)
121    dx, dh0, dWx, dWh, db = rnn_backward(dhv, hv_cache)
122    grads['W_vocab'] = dW_vocab
123    grads['b_vocab'] = db_vocab
124    grads['Wx'] = dWx
125    grads['Wh'] = dWh
126    grads['b'] = db
127    dW_embed = word_embedding_backward(dx, ow_cache)
128    grads['W_embed'] = dW_embed
129    grads['W_proj'] = np.dot(features.T, dh0)
130    grads['b_proj'] = np.sum(dh0, axis=0)
131    return loss, grads

```

第 87 行：定义输入信号 `captions_in`，从正确标注序列中除去最后一个元素后的序列。

第 88 行：定义输出信号 `captions_out`，从正确标注序列中除去第一个元素后的序列。

第 91 行：定义 `mask` 变量，形状为（迷你批次中样本数 N ，标注序列长度 T ），值为布尔型，表示第 i 个样本第 j 个位置是否包含在代价函数计算中。

第 95 行：从属性 `params` 字典中取出参数 `W_proj` 和 `b_proj`，分别为图像特征向量线性变换为隐藏层第 0 时刻输出值时所用连接矩阵和偏移量。

第 98 行：从属性 `params` 字典中取出参数 `W_embed`，通过 `word_embedding_forward` 将 `captions_in` 序列转化为输入层输入信号。

第 101 行：从属性 `params` 字典中取出参数：输入层到隐藏层连接矩阵 `Wx`、隐藏层到隐藏层连接矩阵 `Wh`、隐藏层神经元偏移量 `b`。

第 104 行：从属性 `params` 字典中取出参数：隐藏层到输出层连接矩阵 `W_vocab`、输出层神经元偏移量 `b_vocab`。

第 106 行：定义 `loss` 表示代价函数值，`grads` 字典变量表示对所有参数的导数。

第 107 行：求出迷你批次中样本数 `N`，图像特征向量维度 `D`。

第 108 行：求出标注序列长度 `T`。

第 109 行：求出隐藏层神经元数量 `H`。

第 110 行：将图像特征向量 `features` 参数，经过连接权值矩阵 `Wproj` 和偏移量 `bproj` 的仿射变换，转换为隐藏层第 0 时刻的输出值 `h0`。

第 111 行：调用 `word_embedding_forward` 函数，将输入序列 `captions_in` 通过连接权值矩阵 `W_embed` 输化为输入信号。

第 112 行：定义隐藏层输出值和缓存变量。

第 113~116 行：如果是 RNN 网络，调用 `rnn_forward` 函数求出当前时刻隐藏层输出值 `hv`。

第 117 行：以当前时刻隐藏层输出值 `hv`、隐藏层到输出层连接权值矩阵 `W_vocab`、输出层神经元偏移量 `b_vocab` 为参数，调用 `temporal_affine_forward` 函数，求出输出层输出值。

第 118、119 行：求出输出层代价函数值和对输入值的微分。

第 120 行：调用 `temporal_affine_backward` 函数，将误差反向传播到隐藏层，并求出对隐藏层当前时刻输出的导数 `dhv`、对隐藏层到输出层连接权值的导数 `dW_vocab`、对输出层偏移量的导数 `db_vocab`。

第 121 行：如果是 RNN 网络，调用 `rnn_backward` 函数，将误差传播到输入层，并求出对隐藏层输入信号的导数 `dx`、对前一时刻隐藏层输出的导数 `dh0`、对输入层到隐藏层连接权值的导数 `dWx`、对隐藏层到隐藏层连接权值的导数 `dWh`、隐藏层神经元偏移量的导数 `db`。

第 122~126 行：将已经求出的导数保存到返回值参数求导字典变量中。

第 127 行：调用 `word_embedding_backward` 函数，将误差传递到 `word_embedding` 层，并求出对 `word_embedding` 层到输入层连接权值的导数 `dW_embed`。

第 128~130 行：将求出来的参数导数保存到返回值参数导数字典变量 `grads` 中。

第 131 行：返回代价函数值和参数导数字典。

测试程序入口为：

```
343 if __name__ == '__main__':
344     print('RNN Image Caption Application')
345     test_rnn_loss()
```

运行结果为：

```

RNN Image Caption Application
loss: 9.832356
expected loss: 9.832356
difference: 2.61124455e-12
grads: {'b': array([-1.49088068e-03, -1.16390851e-03, -9.04520440e-04,
-7.00441834e-04, -5.40902949e-04, -4.16799878e-04,
-3.20634060e-04, -2.46343565e-04, -1.89095100e-04,
-1.45075369e-04, -1.11301141e-04, -8.54559494e-05,
-6.57551089e-05, -5.08376733e-05, -3.96828308e-05,
-3.15482259e-05, -2.59283545e-05, -2.25322697e-05,
-2.12812640e-05, -2.23289234e-05, -2.61078864e-05,
-3.34094947e-05, -4.55033658e-05, -6.43014874e-05,
-9.25606755e-05, -1.34088080e-04, -1.93849576e-04,
-2.77758185e-04, -3.91733353e-04, -5.39451346e-04,
-7.18352420e-04, -9.14529047e-04, -1.09949022e-03,
-1.23404533e-03, -1.28234043e-03, -1.22996696e-03,
-1.09231175e-03, -9.05629676e-04, -7.08851657e-04,
-5.29874480e-04]), 'b_proj': array([ 1.54850273e-02, 1.47069033e-02,
1.39287792e-02,
1.31506551e-02, 1.23725310e-02, 1.15944069e-02,
1.08162828e-02, 1.00381587e-02, 9.26003464e-03,
8.48191056e-03, 7.70378647e-03, 6.92566239e-03,
6.14753830e-03, 5.36941421e-03, 4.59129013e-03,
3.81316604e-03, 3.03504195e-03, 2.25691787e-03,

```

从打印结果可以看出，CaptioningRNN.loss 方法返回的代价函数值与正确值之间的误差非常小，所以我们的代码是正确的。由于所有参数的导数的内容太多，所以此处仅列出了开始的一小部分。

下面我们来检查一下计算的各参数的导数的正确性，测试用例代码如下：

```

251 def test_rnn_grads():
252     batch_size = 2
253     timesteps = 3
254     input_dim = 4
255     wordvec_dim = 5
256     hidden_dim = 6
257     word_to_idx = {'<NULL>': 0, 'cat': 2, 'dog': 3}
258     vocab_size = len(word_to_idx)
259
260     captions = np.random.randint(vocab_size, size=(batch_size, timesteps))
261     features = np.random.randn(batch_size, input_dim)
262
263     model = CaptioningRNN(word_to_idx,
264                           input_dim=input_dim,
265                           wordvec_dim=wordvec_dim,
266                           hidden_dim=hidden_dim,
267                           cell_type='rnn',
268                           dtype=np.float64,
269                           )
270
271     loss, grads = model.loss(features, captions)
272
273     for param_name in sorted(grads):
274         f = lambda _: model.loss(features, captions)[0]
275         param_grad_num = eval_numerical_gradient(f, \
276                                                  model.params[param_name], verbose=False, h=1e-6)
277         e = rel_error(param_grad_num, grads[param_name])
278         print('%s relative error: %e' % (param_name, e))
279

```

第 252 行：定义迷你批次中的样本数 N 。

第 253 行：定义标注序列长度 T 。

第 254 行：定义图像特征向量维度 D 。

第 255 行：定义词向量维度 W 。

第 256 行：定义隐藏层神经元数量 H 。

第 257 行：定义单词到索引号字典。

第 258 行：求出单词表中单词数 V 。

第 260 行：生成标注序列和二维数组（迷你批次中的样本数，标注序列长度），对每个元素取 0 到单词数之间的随机数。

第 261 行：用随机数填充图像特征向量。

第 263~269 行：生成 CaptioningRNN 类对象。

第 271 行：调用 CaptioningRNN.loss 方法，求出代价函数值和各参数导数 grads。

第 273 行：遍历 grads 中的各个参数。

第 274~276 行：利用数值计算方法求出对各参数导数的正确值。

第 277 行：求出利用 CaptioningRNN.loss 方法求出的各参数导数与正确值之间的误差。

第 278 行：以科学计数法打印各参数导数误差值。

测试程序入口为：

```
344 if __name__ == '__main__':
345     print('RNN Image Caption Application')
346     test_rnn_grads()
```

运行结果为：

```
RNN Image Caption Application
W_embed relative error: 1.534927e-08
W_proj relative error: 1.847870e-08
W_vocab relative error: 2.666741e-09
Wh relative error: 1.256847e-07
Wx relative error: 3.283662e-06
b relative error: 2.727231e-09
b_proj relative error: 9.052512e-09
b_vocab relative error: 8.667595e-11
```

由上面的测试结果可以看到，通过 CaptioningRNN.loss 方法计算出来的各参数导数与正确值之间的误差非常小，可以认为我们的程序是正确的。

6.2.13 生成图像标记

网络训练完成之后，就可以对给定图片进行图像标注了。图像标注生成方法的测试用例的代码如下：

```
323 def test_sample():
324     data = load_data_ric()
325     small_data = load_coco_data(max_train=50)
326
327     small_rnn_model = CaptioningRNN(
328         cell_type='rnn',
329         word_to_idx=data['word_to_idx'],
330         input_dim=data['train_features'].shape[1],
331         hidden_dim=512,
332         wordvec_dim=256,
333     )
334     for split in ['train', 'val']:
335         minibatch = sample_coco_minibatch(small_data, split=split, \
336                                         batch_size=2)
337         gt_captions, features, urls = minibatch
338         gt_captions = decode_captions(gt_captions, data['idx_to_word'])
339
340         sample_captions = small_rnn_model.sample(features)
341         sample_captions = decode_captions(sample_captions, \
342                                         data['idx_to_word'])
343         print('s:%s' % sample_captions)
```

第 324 行：载入微软图像标注数据集。

第 325 行：取出 50 个样本作为训练样本集。

第 327~333 行：生成 CaptioningRNN 类实例。

第 334 行：将样本分为训练样本集和验证样本集。

第 335、336 行：取出一个迷你批次，包括两个样本。

第 337 行：取出这些样本的标注序列、图像特征向量、图片网址。

第 338 行：将正确的图像标注序列由单词索引号变为真正的单词。

第 340 行：调用 CaptioningRNN.sample 方法，生成新的图像标注序列。

第 341、342 行：将新生成的图像标注序列中单词索引号变为实际的单词。

第 343 行：打印迷你批次中新生成的图像标注信息。

下面我们来看图像标注信息生成函数，代码如下：

```

137 def sample(self, features, max_length=30):
138     """
139     进行一次前向传播过程，根据输入的图像特征向量，生成图像标注信息。
140     在最开始时，将图像特征向量经过W_proj和b_proj仿射变换为隐藏层前一刻的
141     输出值h0，生成只有<START>的标注信息captions，将其经过
142     word_embedding_forward函数以W_embed为参数，将captions变为输入层输入
143     信号ow，将输入信号ow和前一时刻隐藏层输出值，经过rnn_forward函数并以输入
144     层到隐藏层连接权值矩阵Wx，隐藏层到隐藏层连接权值矩阵Wh，隐藏层神经元偏
145     移量b为参数，求出当前时刻隐藏层输出值hv，将当前时刻隐藏层输出值传输到
146     输出层，调用temporal_affine_forward函数，并以隐藏层到输出层连接
147     权值W_vocab和输出层神经元偏移量b_vocab，从输出层输出中，取出概率最大
148     的单词，将其加入到captions中，再将这个新生成的captions重新进行
149     上述操作，直到生成标注序列长度达到最大长度为止。
150     如果是LSTM网络，需要记录细胞的状态，在这种情况下，我们可以高初始状态
151     时细胞状态为0。
152     - features：图像特征向量，形状为（迷你批次中样本数N，特征向量维度D）
153     - max_length：标注标签的最大长度T
154     返回值元组：
155     - captions：每个样本与对应标签的数组，形状为（迷你批次中样本数N，标注标
156       签最大长度），其值为（第i个样本，第j个位置）=单词索引号
157     """
158     N = features.shape[0]
159     captions = self._null * np.ones((N, max_length), dtype=np.int32)
160
161     # Unpack parameters
162     W_proj, b_proj = self.params['W_proj'], self.params['b_proj']
163     W_embed = self.params['W_embed']
164     Wx, Wh, b = self.params['Wx'], self.params['Wh'], self.params['b']
165     W_vocab, b_vocab = self.params['W_vocab'], self.params['b_vocab']
166
167     h0 = np.dot(features, W_proj) + b_proj
168     captions[:, 0] = self._start
169     for n in range(captions.shape[0]):
170         for t in range(1, captions.shape[1]):
171             ow, ow_cache = word_embedding_forward(captions, W_embed)
172             next_h, next_h_cache = rnn_step_forward(ow[:, t, :], \
173                 h0, Wx, Wh, b)
174             h0 = next_h
175             next_h_a = np.zeros((next_h.shape[0], 1, next_h.shape[1]))
176             next_h_a[:, 0, :] = next_h
177             ps, ps_cache = temporal_affine_forward(next_h_a, \
178                 W_vocab, b_vocab)
179             p1 = np.exp(ps[n, 0, :]) / np.sum(np.exp(ps[n, 0, :]))
180             max_val = 0
181             max_idx = -1
182             idx = 0
183             for a11 in p1:
184                 if a11 >= max_val:
185                     max_val = a11
186                     max_idx = idx
187                 idx = idx + 1
188             captions[n, t] = max_idx
189     return captions

```

第 158 行：求出图像特征向量维度 N 。

第 159 行：生成空的图像标签数组 `captions`。

第 162 行：从属性 `params` 字典中取出将图像特征向量转换为隐藏层上一时刻输出值的连接权值矩阵 `W_proj` 和偏移量 `b_proj`。

第 163 行：从属性 `params` 字典中取出标注序列转输入层输入信号时的连接权值矩阵 `W_embed`。

第 164 行：从属性 `params` 字典中取出参数：输入层到隐藏层连接权值矩阵 `Wx`、隐藏层到隐藏层连接权值矩阵 `Wh`、隐藏层神经元偏移量 `b`。

第 165 行：从属性 `params` 字典中取出参数：隐藏层到输出层连接权值矩阵 `W_vocab`、输出层神经元偏移量 `b_vocab`。

第 167 行：将图像特征向量利用连接权值矩阵 `W_proj` 和偏移量 `b_proj` 进行仿射，变换为前一时刻隐藏层输出值 `h0`。

第 168 行：将标注信息 `captions` 的第一个单词设置为 `<start>`。

第 169 行：对每个样本进行循环。

第 170 行：对标注信息序列每个位置进行循环。

第 171 行：将 `captions` 作为输入，调用 `word_embedding_forward` 函数，通过单词嵌入层到输入层连接权值矩阵 `W_embed`，将其转换为输入层输入 `ow`。

第 172、173 行：调用 `rnn_step_forward` 函数，求出隐藏层当前时刻的输出值。

第 174 行：将隐藏层当前时刻状态赋给隐藏层前一时刻状态 `h0`。

第 175~178 行：调用 `temporal_affine_forward` 函数，求出当前样本当前序列位置上，单词表中每个词在该位置上出现的分数（线性输入值）。

第 179 行：利用 `softmax` 函数计算出当前样本当前序列位置上，单词表中每个词在该位置上出现的概率。

第 180~187 行：利用冒泡排序法求出出现概率最大的单词，并将其添加到标注序列的下一个字符位置，循环操作，直至生成整个标注序列为止。

第 188 行：返回标注序列字符串。

测试程序入口为：

```
346 if __name__ == '__main__':
347     print('RNN Image Caption Application')
348     test_sample()
```

运行结果为：

```
RNN Image Caption Application
idx_to_word, <class 'list'>, 1004
val_image_idxs, <class 'numpy.ndarray'>, (195954,), int32
train_captions, <class 'numpy.ndarray'>, (400135, 17), int32
train_features, <class 'numpy.ndarray'>, (82783, 512), float32
train_image_idxs, <class 'numpy.ndarray'>, (400135,), int32
val_features, <class 'numpy.ndarray'>, (40504, 512), float32
val_captions, <class 'numpy.ndarray'>, (195954, 17), int32
train_urls, <class 'numpy.ndarray'>, (82783,), <U63
val_urls, <class 'numpy.ndarray'>, (40504,), <U63
word_to_idx, <class 'dict'>, 1004
s:['<START> outside their jet appliances glasses statue they toys parked assortmen
t dark fire chair skate decker pie toward phone nintendo making candles onto run a
rm decorated held snowboarder blurry refrigerator', '<START> hill containing batte
```

```
r bathtub drinking paper paper laptop colored sunny paper <END>']
s:['<START> buildings background officer toilet it the church seen commercial look
the clean players planes sit brushing sunny apartment path bed go carriage trucks
for colored plant ship', '<START> blanket video colored coffee holds race phone f
amily living water does run is colorful turn for bears ball keyboard phone sunny t
ake house bed chairs drinking lined']
```

6.2.14 网络训练过程

到目前为止，图像标注网络的所有准备工作就都做完了，下面需要做的就是对网络进行训练了。由于图像标注数据集比较大，完全训练需要花费较多的时间，同时为了向大家演示由于样本量过小而出现的过拟合现象，我们将只使用 50 个训练样本组成训练样本集对网络进行训练，用测试样本集进行测试。大家会看到，即使在训练样本集上达到了近乎完美的标注效果，但是在测试样本集上的表现仍然十分糟糕，所以需要我们要加大训练样本集。

网络训练测试程序如下：

```
280 def train_ric():
281     data = load_data_ric()
282     small_data = load_coco_data(max_train=50)
283
284     small_rnn_model = CaptioningRNN(
285         cell_type='rnn',
286         word_to_idx=data['word_to_idx'],
287         input_dim=data['train_features'].shape[1],
288         hidden_dim=512,
289         wordvec_dim=256,
290     )
291     small_rnn_solver = CaptioningSolver(small_rnn_model, small_data,
292         update_rule='adam',
293         num_epochs=50,
294         batch_size=25,
295         optim_config={
296             'learning_rate': 5e-3,
297         },
298         lr_decay=0.95,
299         verbose=True, print_every=10,
300     )
301
302     small_rnn_solver.train()
303
304     # Plot the training losses
305     plt.plot(small_rnn_solver.loss_history)
306     plt.xlabel('Iteration')
307     plt.ylabel('Loss')
308     plt.title('Training loss history')
309     plt.show()
310     for split in ['train', 'val']:
311         minibatch = sample_coco_minibatch(small_data, split=split, \
312             batch_size=2)
313         gt_captions, features, urls = minibatch
314         gt_captions = decode_captions(gt_captions, data['idx_to_word'])
315
316         sample_captions = small_rnn_model.sample(features)
317         sample_captions = decode_captions(sample_captions, \
318             data['idx_to_word'])
319         for gt_caption, sample_caption, url in zip(gt_captions, \
320             sample_captions, urls):
321             plt.imshow(image_from_url(url))
322             plt.title('%s\n%s\nGT:%s' % (split, sample_caption, gt_caption))
323             plt.axis('off')
324             plt.show()
```

第 281 行：装入微软图像标注数据集。

第 282 行：取出 50 个样本作为测试样本集。

第 284~290 行：生成 CaptioningRNN 网络实例。

第 291~300 行：生成 CaptioningSolver 实例，用于管理 CaptionRNN 模型训练过程。

- ❑ `small_rnn_model`: CaptionRNN 实例。
- ❑ `small_data`: 50 个样本的样本集，将被分为训练样本集和验证样本集。
- ❑ `update_rule`: 参数更新规则。
- ❑ `num_epochs`: 样本训练次数，实际迭代次数=总样本数 / 迷你批次中样本数×样本训练次数，例如在当前情况下，实际迭代次数=50/25×50=100。
- ❑ `batch_size`: 迷你批次中样本数。
- ❑ `learning_rate`: 学习率。
- ❑ `lr_decay`: 学习率衰减系数，每次迭代后新学习率=旧学习率×学习率衰减系数。

第 302 行：开始训练过程。

第 305~309 行：当训练结束后，利用 `matplotlib` 绘制横坐标为迭代次数、纵坐标为代价函数值的图形。

第 310 行：对训练样本集和验证样本集进行循环。

第 311、312 行：以迷你批次中样本数为 2，选择测试样本或验证样本，从取出的数据集 `small_data` 中取出组成一个迷你批次的数据。

第 313 行：取出样本的人工标注序列、图像特征向量和图像网址。

第 314 行：将人工标注序列由单词索引号表现形式变为单词本身表现形式。

第 316 行：以图像特征向量为参数，调用 `CaptioningRNN.sample` 方法，求出模型给出的标注序列。

第 317 行：将网络生成标注序列由单词索引号表现形式变为单词本身表现形式。

第 319~324 行：将原始图片从网址处取下来，并显示到界面中，图片标题第一行是网络产生的标注序列，第二行为人工标注序列。

在这个图像标注网络训练中，我们使用了 `CaptioningSolver` 类来具体处理网络训练过程。我们来看一下这个类的代码，类的声明如下：

```
1 import numpy as np
2
3 from cs231n import optim
4 from cs231n.coco_utils import sample_coco_minibatch
5
6
7 class CaptioningSolver(object):
8     """
9     CaptioningSolver 类封装了训练图像标注模型所需要的全部功能，本类使用随机梯度
10    下降算法。
11    本类接收训练和验证样本集，包括图像特征向量和标注序列数据。本类会定期在验证
12    样本集上检查分类精度，避免出现过拟合 (Overfitting) 问题。
13    训练图像标注模型方法：创建图像标注模型类实例，载入微软 coco 图像标注数据集，
14    以此为参数，再加上学习率、迷你批次中样本数、学习率衰减率等，创建本类实例，
15    然后调用本类的 train 方法。
16    当 train 方法结束后，model.params (CaptioningRNN) 字典属性中将包含训练
17    过程中，在验证样本集上表现最好的参数的数值。另外，本类属性 loss_history 中将
18    包含训练过程中代价函数值变化列表，而本类属性 train_acc_history 将包括在每次
19    迭代中在训练样本集和验证样本集上的误差数据。
20    应用举例：
21    data = load_coco_data()
22    model = MyAwesomeModel(hidden_dim=100)
```



```

23 solver = CaptioningSolver(model, data,
24                             update_rule='sgd',
25                             optim_config={
26                                 'learning_rate': 1e-3,
27                             },
28                             lr_decay=0.95,
29                             num_epochs=10, batch_size=100,
30                             print_every=100)
31 solver.train()
32 模型协议
33 传入本类构造函数的模型类必须实现本类规定的模型接口：
34 - model.params：属性必须为字典类型，键为参数名称，值的类型为对应数值类型
35                 numpy的多维数组类型
36 - model.loss(features, captions)：必须具有本方法，用于计算代价函数值和对
37                                 模型中所有参数的导数
38 参数
39 - features：图像特征向量，形状为（迷你批次中样本数N，图像特征向量维度D）
40 - captions：标注序列，形状为（迷你批次中样本数N，标注序列长度T），值为
41             第i个样本第j个位置上出现单词的索引号
42 返回值
43 - loss：代价函数值，为标量
44 - grads：字典类型，对各个参数的导数
45 """
46

```

CaptioningSolver 类的构造函数如下：

```

47 def __init__(self, model, data, **kwargs):
48     """
49     构造函数
50     必需参数
51     - model：符合本类模型协议的类实例，本例中为CaptioningRNN
52     - data：由训练样本集和验证样本集组成的数据集
53     可选参数：
54     - update_rule：学习算法更新规则，默认值为随机梯度下降算法SGD，其他学习
55                   算法定义在optim.py中
56     - optim_config：字典变量，包含选定的学习算法中所需的超参数值。每个学习
57                   算法需要不同的超参数值，请参考optim.py文件定义，但是所有
58                   学习算法均需要学习率超参数，因此必须定义学习率
59     - lr_decay：学习率衰减率，每次迭代后，新学习率=旧学习率*学习率衰减率
60     - batch_size：迷你批次中样本数
61     - num_epochs：循环使用多少次训练样本集
62     - print_every：在多少次循环后打印一次详细信息
63     - verbose：打印更多信息
64     """
65     self.model = model
66     self.data = data
67
68     # Unpack keyword arguments
69     self.update_rule = kwargs.pop('update_rule', 'sgd')
70     self.optim_config = kwargs.pop('optim_config', {})
71     self.lr_decay = kwargs.pop('lr_decay', 1.0)
72     self.batch_size = kwargs.pop('batch_size', 100)
73     self.num_epochs = kwargs.pop('num_epochs', 10)
74
75     self.print_every = kwargs.pop('print_every', 10)
76     self.verbose = kwargs.pop('verbose', True)
77
78     # Throw an error if there are extra keyword arguments
79     if len(kwargs) > 0:
80         extra = ', '.join('%s' % k for k in kwargs.keys())
81         raise ValueError('Unrecognized arguments %s' % extra)
82
83     # Make sure the update rule exists, then replace the string
84     # name with the actual function
85     if not hasattr(optim, self.update_rule):
86         raise ValueError('Invalid update_rule "%s"' % self.update_rule)
87     self.update_rule = getattr(optim, self.update_rule)
88
89     self._reset()
90
91     def _reset(self):
92         """
93         设置属性的默认值，只能在构造函数中调用，在其他地方不能调用
94         """
95

```

```

96     # Set up some variables for book-keeping
97     self.epoch = 0
98     self.best_val_acc = 0
99     self.best_params = {}
100    self.loss_history = []
101    self.train_acc_history = []
102    self.val_acc_history = []
103
104    # Make a deep copy of the optim_config for each parameter
105    self.optim_configs = {}
106    for p in self.model.params:
107        d = {k: v for k, v in self.optim_config.items()}
108        self.optim_configs[p] = d
109
110

```

第 65 行：定义属性 `model`，在本例中为 `CaptioningRNN` 类实例。

第 66 行：定义属性 `data`，为微软 `Coco` 图像标注数据集。

第 69 行：定义属性 `update_rule`，并指定学习算法为随机梯度下降算法 `SGD`。

第 70 行：定义属性 `optim_config`，并从参数中取出值，该值中必须包括学习率超参数。

第 71 行：定义属性 `lr_decay`，并从参数中获取值，默认值为 1.0，表明学习率衰减率。

第 72 行：定义属性 `batch_size`，迷你批次中样本数，从参数中获取，默认值为 100。

第 73 行：定义属性 `num_epochs`，全部训练样本集使用次数，从参数中获取，默认值为 10。

第 75 行：定义属性 `print_every`，隔多少次循环打印一次信息，从参数中获取，默认值为 10。

第 76 行：定义属性 `verbose`，是否打印详细信息，从参数中获取，默认值为 `True`。

第 78~81 行：如果参数中有任何未定义的参数，则抛出异常并退出。

第 83~87 行：确保学习算法在 `optim.py` 中已定义，如果未定义则抛出异常并退出，如果存在则指向实际的学习算法函数。

第 89 行：调用本类私有方法 `_reset`，完成参数的初始化。

第 92 行：定义私有方法 `_reset`。注意，在 `Python` 中，以 “_” 开头的方法和属性全部为私有属性和方法，外部不应该直接访问。

第 97 行：定义属性 `epoch`，记录训练样本集使用过多少次。

第 98 行：定义属性 `best_val_acc`，记录在验证样本集上的最佳精度值。

第 99 行：定义属性 `best_params`，保存在验证样本集上取得最佳精度值时，对应的参数值。

第 100 行：定义属性 `loss_history`，保存代价函数值的历史列表。

第 101 行：定义属性 `train_acc_history`，保存在训练样本集上所求出精度的历史数据。

第 102 行：定义属性 `val_acc_history`，保存在验证样本集上所求出精度的历史数据。

第 104~108 行：将学习算法配置中的超参数复制一份。

下面我们来看 `train` 方法，代码如下：

```

178    def train(self):
179        """
180        神经网络训练程序
181        """
182        num_train = self.data['train_captions'].shape[0]
183        iterations_per_epoch = max(num_train / self.batch_size, 1)
184        num_iterations = int(self.num_epochs * iterations_per_epoch)
185
186        for t in range(num_iterations):
187            self._step()

```

```

188
189     # Maybe print training loss
190     if self.verbose and t % self.print_every == 0:
191         print('(Iteration %d / %d) loss: %f' % (
192             t + 1, num_iterations, self.loss_history[-1]))
193
194     # At the end of every epoch, increment the epoch counter and decay the
195     # learning rate.
196     epoch_end = (t + 1) % iterations_per_epoch == 0
197     if epoch_end:
198         self.epoch += 1
199         for k in self.optim_configs:
200             self.optim_configs[k]['learning_rate'] *= self.lr_decay
201
202     # Check train and val accuracy on the first iteration, the last
203     # iteration, and at the end of each epoch.
204     # TODO: Implement some logic to check Bleu on validation set periodically
205
206     # At the end of training swap the best params into the model
207     self.model.params = self.best_params

```

第 182 行：求出训练样本集中的样本数量。

第 183 行：求出每次遍历训练样本集时的迭代次数，例如训练样本集中包含 100 个训练样本，迷你批次中样本数为 25，则每次遍历训练样本集的迭代次数为 4。

第 184 行：求出总的迭代次数，值为最大允许遍历次数乘以每次遍历的迭代次数，如上例中，假设允许最大遍历次数为 10，则总的迭代次数=10×4=40 次。

第 186 行：总共循环总迭代次数个循环。

第 187 行：第 i 次迭代调用本类私有方法 `_step`，完成训练过程和参数的更新。

第 196 行：判断当前迭代是否为某次遍历的结束点。

第 199、200 行：利用学习率衰减数更新学习算法中的学习率超参数。

第 262~264 行：在每次迭代结束或遍历结束时，分别求出在训练样本集和验证样本集上的误差，如果验证样本集上的误差小于规定的值，则停止训练过程，并与之前最佳验证样本集误差进行比较；如果值持续恶化，则停止训练过程；如果当前达到最佳验证样本集的精度，则记录该精度，并将当前模型参数持久化到模型文件中。这部分内容就留给读者去实现了。

第 207 行：在训练结束时，将最佳参数值赋给模型。

下面我们来看 `_step` 方法，代码如下：

```

111 def _step(self):
112     """
113     由train方法调用，完成一次调用学习算法(以迷你批次为单位)，以对参数导数为
114     条件的参数更新操作，本方法仅限train方法调用，不能直接调用
115     """
116     # Make a minibatch of training data
117     minibatch = sample_coco_minibatch(self.data,
118                                     batch_size=self.batch_size,
119                                     split='train')
120     captions, features, urls = minibatch
121
122     # Compute loss and gradient
123     loss, grads = self.model.loss(features, captions)
124     self.loss_history.append(loss)
125
126     # Perform a parameter update
127     for p, w in self.model.params.items():
128         dw = grads[p]
129         config = self.optim_configs[p]
130         next_w, next_config = self.update_rule(w, dw, config)
131         self.model.params[p] = next_w
132         self.optim_configs[p] = next_config

```

第 117~119 行：从训练样本集中取出相应的迷你批次。

第 120 行：从迷你样本集中取出标注序列、图像特征向量、图片网址信息。

第 123 行：调用模型类（这里指 **CaptioningRNN**）的 `loss` 方法，求出代价函数值和对各参数的导数字典变量。

第 124 行：将代价函数值记录在代价函数值历史列表中。

第 127 行：循环对各参数导数的字典变量。

第 128 行：取出对参数的导数值。

第 129 行：取出学习算法超参数等配置信息。

第 130 行：调用具体学习算法，求出新的参数值和新的学习算法超参数和配置信息。

第 131 行：更新模型中参数值为新求出来的值。

第 132 行：更新学习算法超参数和配置信息。

在 `train` 方法的迭代循环中，为了降低泛化误差，我们需要计算模型在验证样本集上的精度，并且与最佳精度相比较，如果持续恶化则停止训练过程，所以我们需要计算验证样本集上的精度，可以通过调用 `check_accuracy` 方法来实现这个功能，代码如下：

```

135 def check_accuracy(self, X, y, num_samples=None, batch_size=100):
136     """
137     求出在指定样本集（在训练过程中可以为验证样本集或训练样本集，在评价阶段是
138     测试样本集）的分类精度
139     参数
140     - X：所有样本集，为设计矩阵（Design Matrix）
141     - y：分类标签
142     - num_samples：对多少样本进行计算，如果小于总数则向下采样原始样本集
143     - batch_size：迷你批次中样本数
144     返回值
145     - acc：可以正确分类的样本数占总样本数的比例，为标量
146     """
147     # Maybe subsample the data
148     N = X.shape[0]
149     if num_samples is not None and N > num_samples:
150         mask = np.random.choice(N, num_samples)
151         N = num_samples
152         X = X[mask]
153         y = y[mask]
154
155     # Compute predictions in batches
156     num_batches = N / batch_size
157     if N % batch_size != 0:
158         num_batches += 1
159     y_pred = []
160     for i in range(num_batches):
161         start = i * batch_size
162         end = (i + 1) * batch_size
163         scores = self.model.loss(X[start:end])
164         y_pred.append(np.argmax(scores, axis=1))
165     y_pred = np.hstack(y_pred)
166     acc = np.mean(y_pred == y)
167
168     return acc

```

第 148 行：求出样本数量 `N`。

第 149 行：如果 `num_samples` 不为空且小于样本数 `N` 则执行操作。

第 150 行：生成随机向下采样选择掩码。

第 151 行：将样本数重新设为 `num_samples`。

第 152 行：对原始样本集根据选择掩码进行向下采样。

第 153 行：对原始输出标签根据选择掩码进行向下采样。

第 156~158 行：求出迷你批次数量。

第 159 行：模型预测输出值列表。

第 160 行：对每个迷你批次进行循环。

第 161 行：计算当前迷你批次在样本集上的开始位置。

第 162 行：计算当前迷你批次在样本集上的结束位置。

第 163 行：调用模型的 loss 方法，求出代价函数值。

第 164 行：求出代价函数值的最大值，并将其加入模型预测输出列表中。

第 165 行：将其按先后顺序变为一维数组。

第 166 行：求出模型输出正确的样本在总样本中的比例。

测试入口程序为：

```
349 if __name__ == '__main__':
350     print('RNN Image Caption Application')
351     train_ric()
```

运行结果为：

```
RNN Image Caption Application
train_features, <class 'numpy.ndarray'>, (82783, 512), float32
idx_to_word, <class 'list'>, 1004
word_to_idx, <class 'dict'>, 1004
val_captions, <class 'numpy.ndarray'>, (195954, 17), int32
train_image_idx, <class 'numpy.ndarray'>, (400135,), int32
val_features, <class 'numpy.ndarray'>, (40504, 512), float32
train_captions, <class 'numpy.ndarray'>, (400135, 17), int32
val_image_idx, <class 'numpy.ndarray'>, (195954,), int32
val_urls, <class 'numpy.ndarray'>, (40504,), <U63
train_urls, <class 'numpy.ndarray'>, (82783,), <U63
(Iteration 1 / 100) loss: 74.090244
(Iteration 11 / 100) loss: 20.611516
(Iteration 21 / 100) loss: 4.041324
(Iteration 31 / 100) loss: 0.745756
(Iteration 41 / 100) loss: 0.207052
(Iteration 51 / 100) loss: 0.149595
(Iteration 61 / 100) loss: 0.110648
(Iteration 71 / 100) loss: 0.096649
(Iteration 81 / 100) loss: 0.092041
(Iteration 91 / 100) loss: 0.075442
```

将代价函数值与迭代次数的关系绘制成图 6.5，从图中可以看到，由于样本集很小，所以模型很快就收敛了，并且最终的误差非常小。而且，在大约 40 次迭代后，模型在训练样本集上取得了非常好的效果。

如图 6.6，图上第一行文字是模型给出的标注序列信息，第二行文字是图片人工标注的序列信息，但这两行内容几乎完全不符。这是为什么呢？

这就是我们讲到的过拟合的问题。因为训练样本集非常小，新图片与训练样本集上所有图片都不相像是很有可能的。另外，由于我们在训练样本集上得到了万分之几的误差，模型能力远大于训练样本集的学习能力，因此模型为了适应训练样本集，进行了过度优化，而新样本一旦不在训练样本集范围之内，就会出现较大的误差。

那怎么解决这个问题呢？第一种方法是早期停止，就是如果在迭代 35 次左右就停止，训练样本集上的误差也已经比较小，但是还没有出现过拟合现象。另一种方法就是加大训练样本集。关于这个问题，我们将在网络持久化部分来讲解。因为训练样本集加大后，可能一次运行不能完成，需要多次启动。如果我们可以把权值记录下来，下次训练时从上次结束的地方开始，将可以大大缩短训练过程。

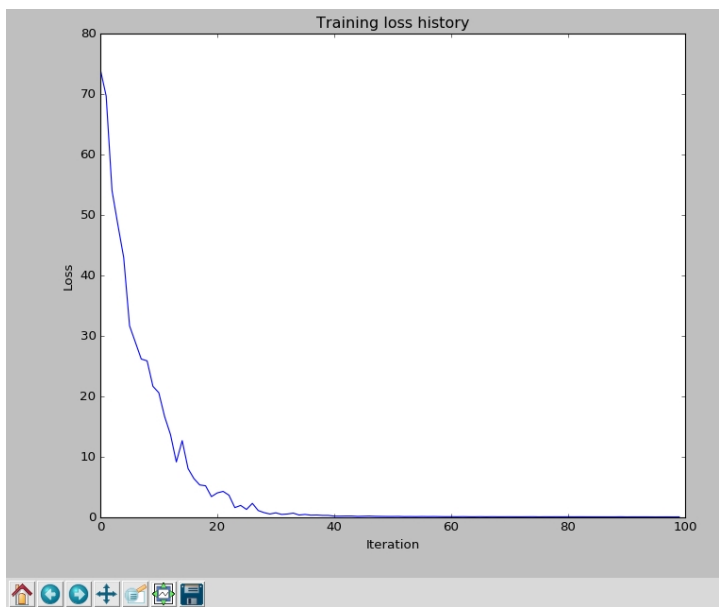


图 6.5 训练样本集上代价函数随迭代次数改变趋势



图 6.6 对新图片进行图像标注

6.2.15 网络持久化

图像标注网络模型训练完成之后就可以直接使用了。但是在当前的实现中，每次要使用时都需要重新训练一遍，若是小样本集这样做还可以勉强接受，但若是大样本集就不太合适了。因此在本节中，我们将讲解如何利用 Python 的 pickle 模块，将模型的参数序列化

到文件中，在网络运行时直接读入模型文件，就可以继续训练或进行实际的图像标注了。

需要说明的是，在 Python 中实际上有两种序列化的方法，一种是在这里要讲的 pickle 模块，另一种是 json 格式。由于 pickle 模块以二进制形式存储，在存储和读取时效率会高一些，但是其很难与其他系统进行信息交换。而 json 格式则相反，它以文本形式存储，效率会降低一些，但是便于系统间交换信息。在这里只需要对模型数据的存储，没有与其他系统进行信息交换的需求，因此选择使用 pickle 模块。

和之前一样，我们先写测试用例，代码如下：

```

348 def test_save_model():
349     data = load_data_ric()
350     small_data = load_coco_data(max_train=5000)
351
352     small_rnn_model = CaptioningRNN(
353         cell_type='rnn',
354         word_to_idx=data['word_to_idx'],
355         input_dim=data['train_features'].shape[1],
356         hidden_dim=512,
357         wordvec_dim=256,
358     )
359     small_rnn_solver = CaptioningSolver(small_rnn_model, small_data,
360         update_rule='adam',
361         num_epochs=10,
362         batch_size=50,
363         optim_config={
364             'learning_rate': 5e-3,
365         },
366         lr_decay=0.95,
367         verbose=True, print_every=10,
368     )
369     model_file = os.getcwd() + '/captioning_rnn.ytm'
370     W_embed = small_rnn_model.params['W_embed']
371     W_embed[0][0] = 8.998
372     print('stored %s' % W_embed)
373     small_rnn_solver.save_model(model_file)

```

第 349 行：读入微软 coco 图像标识数据集。

第 350 行：取出 5000 个样本作为训练样本集。

第 352～358 行：生成 CaptioningRNN 类实例。

第 359～368 行：生成 CaptioningSolver 实例

第 369 行：定义模型文件全路径文件名。

第 370 行：取出模型中单词嵌入连接权值矩阵。

第 371 行：修改第一个元素为 8.998。

第 372 行：打印 W_embed 参数内容，确认进行了修改。

第 373 行：调用 CaptioningSolver.save_model 方法，将模型参数保存到模型文件中。

CaptioningSolver 类中 save_model 方法的实现如下：

```

204 def save_model(self, model_file):
205     """
206     将模型的参数保存到指定的模型文件中
207     """
208     print('save model parameters')
209     mf_obj = open(model_file, 'wb')
210     pickle.dump(self.model.params, mf_obj)
211     mf_obj.close()

```

第 204 行：定义保存模型参数的方法，model_file 为模型文件全路径名，并且必须确保对这个文件有写的权限。

第 209 行：以二进制写的方式打开模型文件。

第 210 行：将模型参数字典对象序列化保存到模型文件中。

第 211 行：关闭模型文件。

测试程序入口为：

```
401 if __name__ == '__main__':
402     print('RNN Image Caption Application')
403     test_save_model()
```

运行结果为：

```
RNN Image Caption Application
train_captions, <class 'numpy.ndarray'>, (400135, 17), int32
idx_to_word, <class 'list'>, 1004
word_to_idx, <class 'dict'>, 1004
train_urls, <class 'numpy.ndarray'>, (82783,), <U63
val_features, <class 'numpy.ndarray'>, (40504, 512), float32
train_image_idx, <class 'numpy.ndarray'>, (400135,), int32
train_features, <class 'numpy.ndarray'>, (82783, 512), float32
val_image_idx, <class 'numpy.ndarray'>, (195954,), int32
val_urls, <class 'numpy.ndarray'>, (40504,), <U63
val_captions, <class 'numpy.ndarray'>, (195954, 17), int32
stored [[ 8.99800014e+00 -7.57191190e-03 -2.04756018e-02 ...,  1.09494459e-02
-3.92883364e-03  1.00661265e-02]
 [ 2.32477381e-04  1.45973880e-02 -1.89327542e-03 ..., -9.30197909e-03
 8.12873710e-03  1.72856613e-03]
 [ -6.76558772e-03  6.81014638e-03  9.17418487e-03 ..., -6.61994284e-03
 1.23539185e-02 -1.79777816e-02]
 ...,
 [ 1.12784747e-02  1.36185540e-02 -1.82536505e-02 ...,  1.90455019e-02
-4.93589276e-03  1.09900227e-02]
 [ 5.33971761e-04 -5.74123533e-03  4.93653992e-04 ..., -2.70833578e-02
-1.70940068e-02 -5.90339815e-03]
 [ 6.44608866e-03  2.04271544e-03 -7.82560091e-03 ..., -2.62113404e-03
-1.67429994e-03 -5.43343369e-04]]
save model paramters
```

通常我们需要在模型训练结束时保存模型参数，将模型参数保存到指定的模型文件中。

从模型文件中恢复模型参数的过程如下：

```
375 def test_restore_model():
376     data = load_data_ric()
377     small_data = load_coco_data(max_train=5000)
378
379     small_rnn_model = CaptioningRNN(
380         cell_type='rnn',
381         word_to_idx=data['word_to_idx'],
382         input_dim=data['train_features'].shape[1],
383         hidden_dim=512,
384         wordvec_dim=256,
385     )
386     small_rnn_solver = CaptioningSolver(small_rnn_model, small_data,
387         update_rule='adam',
388         num_epochs=10,
389         batch_size=50,
390         optim_config={
391             'learning_rate': 5e-3,
392         },
393         lr_decay=0.95,
394         verbose=True, print_every=10,
395     )
396     model_file = os.getcwd() + '/captioning_rnn.ytm'
397     small_rnn_solver.restore_model(model_file)
398     W_embed = small_rnn_model.params['W_embed']
399     print('restored %s' % W_embed)
```

第 376 行：读入微软 coco 图像标识数据集。

第 377 行：取出 5000 个样本作为训练样本集。

第 379~385 行：生成 CaptioningRNN 类实例。

第 386~395 行：生成 CaptioningSolver 实例。

第 396 行：定义模型文件全路径文件名。

第 397 行：调用 CaptioningSolver.restore_model 方法，读出模型参数信息，并赋给模型参数字典属性。

第 398 行：取出单词嵌入连接权值矩阵 W_embed。

第 399 行：打印 W_embed 参数内容，确认第一个元素的值为 8.998。

CaptioningSolver 类中 restore_model 方法的实现如下：

```
213 def restore_model(self, model_file):
214     """
215     从模型文件中读出参数内容并赋给网络模型
216     """
217     print('restore model from file: %s' % model_file)
218     mf_obj = open(model_file, 'rb')
219     self.model.params = pickle.load(mf_obj)
220     mf_obj.close()
```

第 213 行：定义 restore_model 方法，model_file 为模型文件名。

第 218 行：以二进制只读方式打开模型文件。

第 219 行：读出模型文件中内容并赋给模型参数字典属性 params。

第 220 行：关闭模型文件。

测试入口程序为：

```
401 if __name__ == '__main__':
402     print('RNN Image Caption Application')
403     test_save_model()
```

运行结果为：

```
RNN Image Caption Application
val_urls, <class 'numpy.ndarray'>, (40504,), <U63
train_image_idx, <class 'numpy.ndarray'>, (400135,), int32
val_captions, <class 'numpy.ndarray'>, (195954, 17), int32
train_features, <class 'numpy.ndarray'>, (82783, 512), float32
train_urls, <class 'numpy.ndarray'>, (82783,), <U63
word_to_idx, <class 'dict'>, 1004
idx_to_word, <class 'list'>, 1004
train_captions, <class 'numpy.ndarray'>, (400135, 17), int32
val_features, <class 'numpy.ndarray'>, (40504, 512), float32
val_image_idx, <class 'numpy.ndarray'>, (195954,), int32
restore model from file: /home/osboxes/dev/wky/work/book/chp06/e002/captioning_rnn.y
tm
restored [[ 8.99800014e+00 -1.19222198e-02 -7.45383138e-03 ..., -8.30541551e-03
 1.73401572e-02  8.47334974e-03]
 [ -1.75167341e-03 -6.49535423e-03 -2.13973224e-02 ...,  3.25383875e-03
  8.45439546e-03 -1.37866223e-02]
 [  4.28727828e-03  5.09749679e-03 -1.10880742e-02 ...,  1.40188290e-02
  3.42482817e-03 -9.38608311e-03]
 ...,
 [ -1.60019491e-02 -2.40409840e-03  5.87148499e-03 ...,  3.48441629e-03
 -8.00677016e-03 -4.61746426e-03]
 [ -8.52204673e-03 -1.61863733e-02 -7.81052094e-03 ...,  1.03308512e-02
  3.23473988e-03  2.70170887e-04]
 [  1.43864630e-02  1.13894958e-02  2.48558377e-03 ..., -1.91875398e-02
 -1.20953545e-02 -2.33898535e-02]]
```

由打印内容可以看出，单词嵌入连接权值变量 W_embed 的第一项值为 8.998。注意，浮点数在计算机表示中是有误差的。

第 7 章

长短时记忆网络

上一章讲述了递归神经网络的基本概念，并且以计算机写作和图像标注为例，讲解了普通递归神经网络的应用场景。但是如果读者留意的话，会发现目前很少有人应用普通递归神经网络了，因为它会出现梯度消失现象。为了解决这个问题，20 世纪 90 年代，出现过很多改良的递归神经网络模型，其中最成功的当属长短时记忆网络。

在本章中，首先讲述普通递归神经网络的缺陷，然后引入典型的长短时记忆网络。接着会以情感计算为例，讲解怎样使用 Theano 框架，采用大型影评数据库 IMDB 为训练样本集，进行影评内容的情感计算。

7.1 长短时记忆网络原理

7.1.1 网络架构

虽然标准的递归神经网络可以在一定的上下文环境中，根据输入序列得出输出序列，但是其可以获取的上下文信息是有限的，这就制约了它被广泛采用。因为在使用过程中，普通递归神经网络早期输入信号对隐藏层是有影响的，一方面可能会逐渐衰减；另一方面可能会逐渐增强直至出现溢出，即梯度消失问题。下面我们以输入信号影响隐藏层衰减为例，来看这一现象出现的原因，如图 7.1 所示。

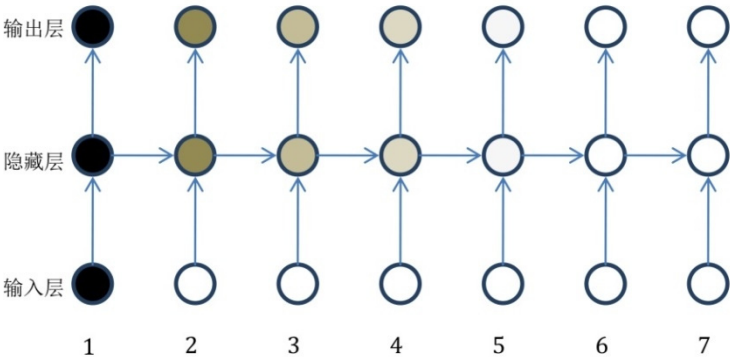


图 7.1 RNN 输入信号影响隐藏层衰减

图中 $t=1$ 时刻的输入信号为 $x^{(1)}$ ，此时对隐藏层和输出层的影响最大；当 $t=2$ 时刻时，输入信号 $x^{(1)}$ 对隐藏层和输出层的影响将减小一些；到 $t=3$ 时刻时，输入信号 $x^{(1)}$ 对隐藏层和输出层的影响将更小；在 $t=6$ 时刻时，输入信号 $x^{(1)}$ 对隐藏层和输出层的影响基本可以忽略不计了。输入信号逐渐加强的原理与此类似。

为了解决这个问题，人们提出了各种改进的递归神经网络模型，有些研究人员提出不使用梯度下降算法（如模拟退火法和离散误差传播法），而采用时延方式等。在这些改进模型中，长短时记忆网络是最成功的一种，下面将主要讲解这一模型。

长短时记忆网络由一系列递归连接的记忆区块的子网络构成，这些区块可以视为计算机中可微版的内存芯片。每个记忆区块中包含一个或多个记忆细胞和三个乘法单元：输入门、输出门和遗忘门，可以对记忆细胞进行连续的写、读和重置操作。

典型的长短时记忆网络结构如图 7.2 所示。

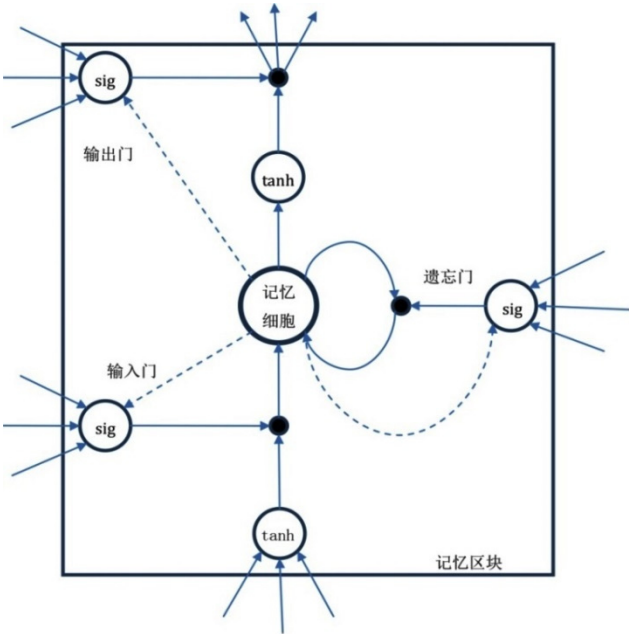


图 7.2 典型的长短时记忆网络架构

这是一个典型的长短时记忆网络的记忆区块架构图，图中包含一个记忆细胞。输入门、输出门和遗忘门是三个非线性累加单元，分别累加区块内和区块外的信号，通过黑色圆圈代表的乘法单元，控制记忆细胞的激活。输入门与输入信号相乘，输入到记忆细胞中，可以通过控制输入门的开闭控制是否接收输入信号。输出门与输出信号相乘，通过控制输出门开闭，可以控制何时将输出信号输出到输出层。遗忘门与前一状态连接，通过控制遗忘门的开闭，可以控制是否遗忘之前的输入信号。以上各个连接均没有连接权值，或者可以认为连接权值为 1。图中记忆细胞和三个门之间由虚线连接，被称为窥视孔连接，是具有连接权值的。通常，输入门、输出门、遗忘门使用 Sigmoid 激活函数，而输入和记忆细胞输出采用双曲正切函数。

下面我们来分析一下，为什么长短时记忆网络可以解决梯度消失问题。我们采用递归神经网络的展开形式来讲解这一问题，如图 7.3 所示。

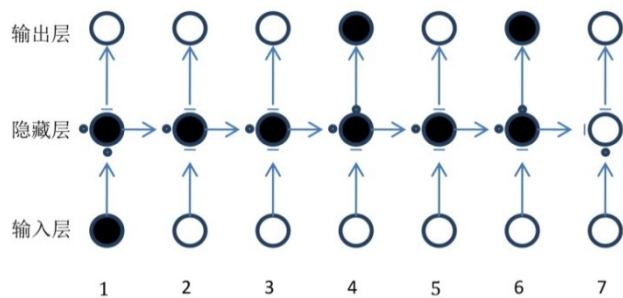


图 7.3 长短时记忆网络的展开图

图中的小圆圈、短横线或短竖线分别代表三个门，其中左边的代表遗忘门，下边的代表输入门，上边的代表输出门，小圆圈代表打开状态，短横线或短竖线代表关闭状态。

在 $t=1$ 时刻：输入门和遗忘门处于打开状态，输出门处于关闭状态，输入信号进入记忆区块。

在 $t=2$ 时刻：输入门和输出门关闭，遗忘门打开，因此前一时刻的状态可以影响当前时刻的记忆区块。

在 $t=3$ 时刻：输入门和输出门关闭，遗忘门打开，因此前一时刻的状态可以影响当前时刻的记忆区块，与普通递归神经网络不同的是，这时输入信号对隐藏层的影响并没有衰减。

在 $t=4$ 时刻：输入门处于关闭状态，遗忘门和输出门处于开启状态，此时记忆区块的输出会传输到输出层。

在 $t=5$ 时刻，输入门和输出门关闭，遗忘门打开，此时不接收新的输入信号，前一时刻的状态会影响当前时刻的记忆区块。

在 $t=6$ 时刻，输入门处于关闭状态，遗忘门和输出门处于打开状态，上一时刻的记忆区块传递到当前时刻的记忆区块，同时记忆区块输出会传输到输出层。

在 $t=7$ 时刻，输入门处于打开状态，遗忘门和输出门处于关闭状态，上一时刻的记忆区块状态将被遗忘，同时可以接收新的输入信号。

由此可见，通过控制输入门的开启和关闭，可以控制新的输入信号进入记忆区块，从

而解决普通递归神经网络中输入信号衰减的问题。通过控制遗忘门开启和关闭，可以控制最初的输入信号起作用的范围，从而防止出现非常多的输入信号叠加，造成溢出问题。通过控制输出门的开启和关闭，可以控制输入信号在什么时间对网络输出产生影响。长短期记忆网络正是通过学习算法，以及精确控制输入门、遗忘门、输出门的开启与关闭，来解决很多普通递归神经网络不能解决的序列问题。

我们目前讨论的只是单个记忆区块，实际上，记忆区块是作为递归神经网络隐藏层节点来组成网络的，如图 7.4 所示。

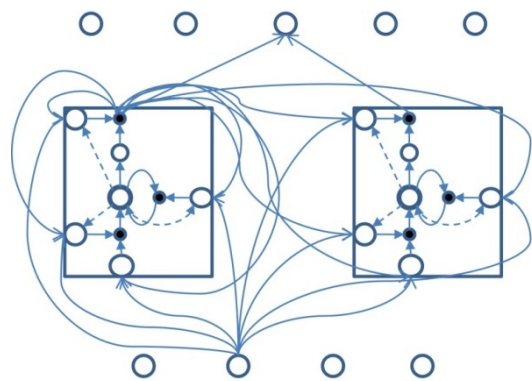


图 7.4 长短期记忆网络的架构

图中输入层有 4 个神经元，隐藏层由 2 个记忆区块组成，输出层有 5 个神经元。为了显示更清晰，我们只画出了输入层的一个神经元和输出层的一个神经元。

在实际应用中，长短期记忆网络在一系列需要长期记忆的合成任务中取得了巨大的成功。因为只要输入门关闭，新来的输入信号就不会覆盖原来的输入信号，通过打开输出门，可以使原来的输入信号产生作用。这些机制使长短期记忆网络可以解决很多普通递归神经网络不能解决的问题。

长短期记忆网络已经成功应用于上下文无关语言学习、在含有噪声的实数序列中回忆曾经出现的实数序列。另外，长短期记忆网络在蛋白质二级结构、音乐生成、增强学习、语音识别、手写字体识别等领域都取得了巨大的成功。

7.1.2 数学公式

假设输入层维度为 D ，在 t 时刻输入信号为： $\boldsymbol{x}^{(t)} = \mathbb{R}^D$ 。

假设隐藏层神经元个数为 H ，上一时刻 $(t-1)$ 隐藏层状态用隐藏层输出值表示为： $\boldsymbol{a}^{(2)(t-1)} = \mathbb{R}^H$ 。在长短期记忆网络中，还需要维护记忆细胞状态为： $\boldsymbol{c}^{(t-1)} \in \mathbb{R}^H$ 。

对于输入门、遗忘门、输出门、区块输入来说，输入均由两部分组成，即记忆区块的输入信号和上一时刻隐藏层状态（输出信号）。记忆区块输入信号维度为 D ，组织方式为：输入门有 H 个单元，上一时刻隐藏层状态（输出信号）维度为 H 。其输入可以定义为如图 7.5 所示结构。

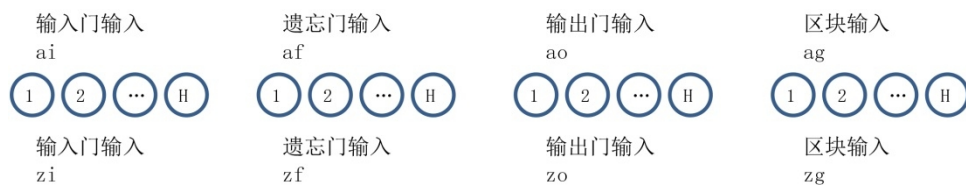


图 7.5 记忆区块组织结构

可以将其视为具有 $4H$ 个神经元的隐藏层，所以输入层到记忆区块的连接权值矩阵为： $\mathbf{W}^{(1,2)} \in \mathbb{R}^{4H \times D}$ 。同理，上一时刻隐藏层状态（输出信号）与当前隐藏层的连接权值矩阵为： $\mathbf{W}^{(2,2)} \in \mathbb{R}^{4H \times H}$ 。

下面分别来看各个逻辑单元信号处理过程，首先定义输入信号：

$$\mathbf{x}^{(t)} = \mathbf{a}^{(1)(t)}$$

上一时刻隐藏层状态（输出信号）定义为： $\mathbf{a}^{(2)(t-1)}$ 。

对于输入门：

$$\text{输入: } \mathbf{z_i}^{(2)(t)} = (\mathbf{W}^{(1,2)} \mathbf{a}^{(1)(t)} + \mathbf{W}^{(2,2)} \mathbf{a}^{(2)(t-1)} + \mathbf{b}^{(2)})[1:H]$$

$$\text{输出: } \mathbf{a_i}^{(2)(t)} = \text{sigmoid}(\mathbf{z_i}^{(2)(t)})$$

对于遗忘门：

$$\text{输入: } \mathbf{z_f}^{(2)(t)} = (\mathbf{W}^{(1,2)} \mathbf{a}^{(1)(t)} + \mathbf{W}^{(2,2)} \mathbf{a}^{(2)(t-1)} + \mathbf{b}^{(2)}) \left[H + \frac{1}{2}H \right]$$

$$\text{输出: } \mathbf{a_f}^{(2)(t)} = \text{sigmoid}(\mathbf{z_f}^{(2)(t)})$$

对于输出门：

$$\text{输入: } \mathbf{z_o}^{(2)(t)} = (\mathbf{W}^{(1,2)} \mathbf{a}^{(1)(t)} + \mathbf{W}^{(2,2)} \mathbf{a}^{(2)(t-1)} + \mathbf{b}^{(2)}) \left[2H + \frac{1}{3}H \right]$$

$$\text{输出: } \mathbf{a_o}^{(2)(t)} = \text{sigmoid}(\mathbf{z_o}^{(2)(t)})$$

对于区块输入：

$$\text{输入: } \mathbf{z_g}^{(2)(t)} = (\mathbf{W}^{(1,2)} \mathbf{a}^{(1)(t)} + \mathbf{W}^{(2,2)} \mathbf{a}^{(2)(t-1)} + \mathbf{b}^{(2)}) \left[3H + \frac{1}{4}H \right]$$

$$\text{输出: } \mathbf{a_g}^{(2)(t)} = \tanh(\mathbf{z_g}^{(2)(t)})$$

下面来计算记忆细胞本时刻状态：

$$\mathbf{c}^{(2)(t)} = \mathbf{z_i}^{(2)(t)} \odot \mathbf{z_g}^{(2)(t)} + \mathbf{c}^{(t-1)} \odot \mathbf{z_f}^{(2)(t)} \quad \odot \text{代表点积操作}$$

下面来求记忆区块输出信号：

$$\mathbf{a}^{(2)(t)} = \mathbf{z_o}^{(2)(t)} \odot \tanh(\mathbf{c}^{(2)(t)})$$

以上是信号的前向传播过程，误差反向传播等过程与传统递归神经网络类似，我们将在下面的例子中详细讲解。

7.2 MNIST 手写数字识别

在实际应用中，直接使用普通递归神经网络的情况比较少，在大多数情况下，使用的均是长短时记忆网络，最成功的应用就是语间识别和自然语言处理方面。但是实际上，长短时记忆网络在图像处理、时序信号处理（如股票价格预测）等方面也有非常成功的应用，只不过大家关注较少而已。在这一章里，将向大家介绍长短时记忆网络在 MNIST 手写数字识别领域的应用，将使用 TensorFlow 框架来实现这一功能。

首先来看数据集的读入，将以行为单位，将图像中的一行像素值作为网络的输入信号，因此需要对 TensorFlow 读入的 MNIST 手写数字识别数据集进行预处理，变为以行为单位进行存储，代码如下：

```
1 def load_datasets(self):
2     mnist = input_data.read_data_sets(self.datasets_dir,
3         one_hot=True)
4     raw_X_train = mnist.train.images
5     X_train = raw_X_train.reshape(-1, self.input_vec_size, self.input_vec_size)
6     y_train = mnist.train.labels
7     raw_X_validation = mnist.validation.images
8     X_validation = raw_X_validation.reshape(-1, self.input_vec_size, self.input_vec_size)
9     y_validation = mnist.validation.labels
10    raw_X_test = mnist.test.images
11    X_test = raw_X_test.reshape(-1, self.input_vec_size, self.input_vec_size)
12    y_test = mnist.test.labels
13    return X_train, y_train, X_validation, y_validation, \
14        X_test, y_test, mnist
```

第 2、3 行：调用 TensorFlow 的 `input_data` 的 `read_data_sets` 方法，第一个参数为数据集存放路径，第二个参数是标签集的格式。在原始 MNIST 数据集中，我们知道每个样本是 28×28 的黑白图片，对应的是 0~9 的数字标签，所以其格式为：[...784 (28×28) 像素点的值...][3]。其中，第一项为 784 (28×28) 个 0~1 的浮点数，0 代表黑色，1 代表白色；第二项的“3”代表这个样本是数字 3。为了后续处理方便，我们将标签[3]改为 one-hot 形式，因为标签代表 0~9 的数字，所以标签集为 10 维向量，每维上取值为 0 代表不是这个对应位置的数字，取值为 1 代表是这个对应位置的数字。其中，只有一维可以取 1，因此称之为 one-hot，还以上面的例子为例，标签集的格式就变为：[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]，因为第四位为 1，所以代表这个样本是数字 3。

第 4 行：取出训练样本集输入信号集 `X_train`，其为设计矩阵形式，每一行代表一个样本，行数为训练样本集中的样本数量。在这个例子中，就 $X_{train} \in \mathbb{R}^{55000 \times 784}$ ，其中训练样本集中有 55000 个样本，每个样本是 784 (28×28) 维的图片。

第 5 行：将样本形状由 784 变为 28×28 ，因为我们将以行为单位，将图像样本数据输入长短时记忆网络中。我们每次输入 1 行（28 个像素值），作为一个时间 t 的输入，一幅图像共有 28 行，分为 28 个时间点顺序输入。

第 6 行：取出训练样本集标签集 `y_train`，其为 one-hot 为行的矩阵，每一行代表对应样

本的正确结果，行数为训练样本集中的样本数量。在这个例子中，就 $y_{\text{train}} \in \mathbb{R}^{55000 \times 10}$ ，其中训练样本集中有 55000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 7 行：取出验证样本集输入信号集 $X_{\text{validation}}$ ，其为设计矩阵形式，每一行代表一个样本，行数为验证样本集中的样本数量。在这个例子中，就 $X_{\text{validation}} \in \mathbb{R}^{5000 \times 784}$ ，其中验证样本集中有 5000 个样本，每个样本是 784 (28×28) 维的图片。根据前面我们的讨论可以知道，在训练过程中，为了防止模型出现过拟合，模型的泛化能力降低（模型在训练样本集达到非常高的精度，但是在未见过的测试样本集或实际应用中，精度反而不高），通常会采用 Early Stopping 策略，就是在逻辑回归模型训练过程中，只用训练样本集对模型进行训练，每隔一定的时间间隔，计算模型在未见过的验证样本集上识别的精度，并记录迄今为止在验证样本集上取得的最高精度。我们会发现，在训练初期，验证样本集上的识别精度会稳步提高，但是到了一定阶段之后，验证样本集上的识别精度就不会再明显提高了，甚至开始逐渐下降，这就说明模型出现了过拟合，这时就可以停止模型训练，将在验证样本集上取得最佳识别精度的模型参数作为模型最终的参数。综上所述，验证样本集主要用于防止模型出现过拟合，为 Early Stopping 算法提供终止依据。

第 8 行：将样本形状由 784 变为 28×28，因为我们将以行为单位将图像样本数据输入长短时记忆网络中。我们每次输入 1 行（28 个像素值），作为一个时间 t 的输入，一幅图像共有 28 行，分为 28 个时间点顺序输入。

第 9 行：取出验证样本集标签集 $y_{\text{validation}}$ ，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为训练样本集中的样本数量。在这个例子中，就 $y_{\text{validation}} \in \mathbb{R}^{5000 \times 10}$ ，其中验证样本集中有 5000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 10 行：取出测试样本集输入信号集 X_{test} ，其为设计矩阵形式，每一行代表一个样本，行数为训练样本集中的样本数量。在这个例子中，就 $X_{\text{test}} \in \mathbb{R}^{10000 \times 784}$ ，其中训练样本集中有 10000 个样本，每个样本是 784 (28×28) 维的图片。测试样本集主要用于模型训练结束后对模型性能进行评估。由于模型没有见过测试样本集中的样本，可以模拟模型在实际部署之后的情况，模型在测试样本集上的识别精度，基本可以视为模型在实际应用中可以达到的精度。

第 11 行：将样本形状由 784 变为 28×28，因为我们将以行为单位，将图像样本数据输入长短时记忆网络中。我们每次输入 1 行（28 个像素值），作为一个时间 t 的输入，一幅图像共有 28 行，分为 28 个时间点顺序输入。

第 12 行：取出测试样本集标签集 y_{test} ，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为测试样本集中的样本数量。在这个例子中，就 $y_{\text{test}} \in \mathbb{R}^{10000 \times 10}$ ，其中测试样本集中有 10000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 13、14 行：返回训练样本集输入信号集、训练样本集标签集、验证样本集输入信号集、验证样本集标签集、测试样本集输入信号集、测试样本集标签集。

在讲述模型创建过程之前，先来看一下 Lstm_Engine 类的构造函数，代码如下：


```

1 def __init__(self):
2     self.datasets_dir = 'datasets/'
3     self.input_vec_size = self.lstm_size = 28      # 输入向量的维度
4     self.time_step_size = 28                      # 循环层长度
5     self.batch_size = 128
6     self.test_size = 256
7     self.num_category = 10

```

第 2 行：定义数据集文件存放目录。

第 3 行：定义输入信号维度为 28，即图像每行的像素数，同时定义隐藏层 LSTM Cell 数目为 28。

第 4 行：定义时间步数，由于图像为 28×28，共有 28 行，所以步数设置为 28，可以完整地处理完一幅图像。

第 5 行：训练样本集上的迷你批次大小为 128。

第 6 行：测试样本集上的迷你样本集大小为 256。

第 7 行：预分类类别为 10，即 0~9 共 10 个数字。

下面我们来看模型创建过程，代码如下：

```

1 def build_model(self):
2     self.X = tf.placeholder(tf.float32, [None, self.time_step_size, self.input_vec_size])
3     XT = tf.transpose(self.X, [1, 0, 2])
4     XR = tf.reshape(XT, [-1, self.lstm_size])
5     X_split = tf.split(XR, self.time_step_size, 0)
6     self.y = tf.placeholder("float", [None, self.num_category])
7     W = tf.Variable(tf.random_normal(shape=[self.lstm_size, self.num_category],
8     mean=0.0, stddev=0.01))
9     b = tf.Variable(tf.random_normal(shape=[self.num_category], mean=0.0,
10    stddev=0.0001))
11    lstm = tf.contrib.rnn.BasicLSTMCell(self.lstm_size, forget_bias=1.0, state_is_tuple=True)
12    z, _states = tf.contrib.rnn.static_rnn(lstm, X_split, dtype=tf.float32)
13    y_ = tf.matmul(z[-1], W) + b
14    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=y_, labels=self.y))
15    self.train_op = tf.train.RMSPropOptimizer(0.001, 0.9).minimize(cost)
16    self.correct_prediction = tf.equal(tf.argmax(y_, 1), tf.argmax(self.y, 1))
17    self.accuracy = tf.reduce_mean(tf.cast(self.correct_prediction, tf.float32))
18    self.y_ = y_

```

第 2 行：创建输入信号的 placeholder，输入信号维度为每行像素数 28，共分 time_step_size=28 步输入。第 1 维为 None，代表迷你批次中的序号。其格式为：（迷你批次数号，时间序号，输入向量）。

第 3 行：将第 1 维和第 2 维互换，变为：（时间序号，迷你批次数号，输入向量）格式。

第 4、5 行：将上步的三维数组拆分为多个二维数组，每个二维数组代表一个时间点，为一个迷你批次在该时间点的所有输入信号。从这里可以看出，我们实际上是将 128（迷你批次大小）个图像的第 1 行像素作为第 1 个时间点，先输入到隐藏层 LSTM Cell 中，然后输入第 2 个时间点的第 2 行像素，以此类推，经过 28（time_step_size=28）个时间点，完成本迷你批次中 128 个图像数据的输入。

第 6 行：定义存放分类正确结果的 placeholder。

第 7、8 行：定义隐藏层 LSTM Cell 和输出层之间的连接权值矩阵，采用均值为 0.0、标准差为 0.01 的正态分布随机数进行初始化。

第 9、10 行：定义输出层的偏置值，采用均值为 0.0、标准差为 0.0001 的正态分布随机数进行初始化。

第 11 行：定义 LSTM Cell。TensorFlow 对 LSTM Cell 进行了封装，可以直接使用，不需要考虑输入门、遗忘门、输出门和输入调制门的实现细节，实际其定义了如图 7.6 所示结构的 LSTM Cell。

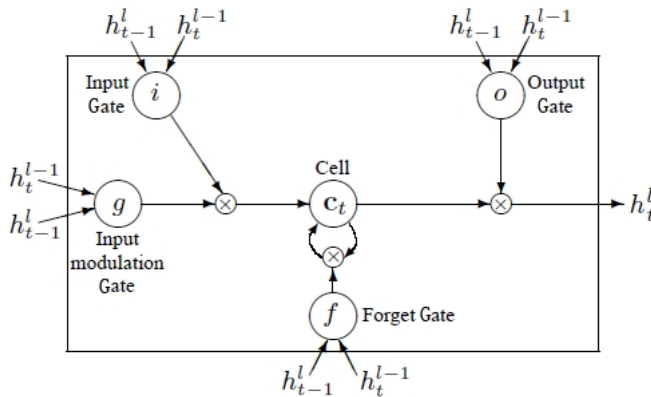


图 7.6 LSTM Cell 结构

上图显示的是多层 LSTM Cell 的形式，在单层的情况下，可以将 h_t^{l-1} 视为网络输入信号。

第 12 行：经过 LSTM Cell 的计算，得到本层输出信号，与网络输入信号相对应，输出信号也以时间点为单位，每个时间点为迷你批次中全部（128 个）样本的输出向量。

第 13 行：求出输出层的输入信号。

第 14 行：使用 softmax 激活函数，与正确分类结果进行比对，求出二者之间的交叉熵，作为网络的代价函数。

第 15 行：采用 RMSProp 优化方法，求代价函数的最小值。

第 16 行：利用 TensorFlow 的 argmax 函数，分别求出计算类别向量每个样本的最大值下标和类别向量每个样本的最大值下标，并对其进行比较。

第 17 行：求出预测精度，先调用 TensorFlow 的 cast 函数，将第 16 行的结果变为浮点数列表，格式为：[1.0, 1.0, 0.0, 1.0, 1.0]，这里假设只有 5 个样本。再调用 TensorFlow 的 reduce_mean 函数求出这个列表的平均值：(1.0+1.0+0.0+1.0+1.0)/5=0.8，这个值就是模型预测的精度。

第 18 行：将计算出的分类类别 y_ 保存为本类属性。

下面我们来看模型训练方法：

```
1 def train(self, mode=TRAIN_MODE_NEW, ckpt_file='work/lgr.ckpt'):
2     X_train, y_train, X_validation, y_validation, X_test, \
3         y_test, mnist = self.load_datasets()
```

```

4     self.build_model()
5     epochs = 100
6     saver = tf.train.Saver()
7     check_interval = 50 # 50
8     best_accuracy = -0.01
9     improve_threthold = 1.005
10    no_improve_steps = 0
11    max_no_improve_steps = 200 #3000
12    is_early_stop = False
13    eval_runs = 0
14    eval_times = []
15    train_accs = []
16    validation_accs = []
17    with tf.Session() as sess:
18        sess.run(tf.global_variables_initializer())
19        if Lstm_Engine.TRAIN_MODE_CONTINUE == mode:
20            saver.restore(sess, ckpt_file)
21        for epoch in range(epochs):
22            if is_early_stop:
23                break
24            batch_idx = 1
25            for start, end in zip(range(0, len(X_train), self.batch_size),
26                                range(self.batch_size, len(X_train)+1,
27                                    self.batch_size)):
28                if no_improve_steps >= max_no_improve_steps:
29                    is_early_stop = True
30                    break
31                sess.run(self.train_op, feed_dict={self.X:
32                                                    X_train[start:end],
33                                                    self.y: y_train[start:end]})
34            no_improve_steps += 1
35            if batch_idx % check_interval == 0:
36                eval_runs += 1
37                eval_times.append(eval_runs)
38                train_accuracy = sess.run(self.accuracy,
39                                          feed_dict={self.X: X_train,
40                                                      self.y: y_train})
41                #train_accuracy = self.calculate_accuracy(sess,
42                #                                           X_train, y_train)
43                train_accs.append(train_accuracy)
44                validation_accuracy = sess.run(self.accuracy,
45                                               feed_dict={self.X: X_validation,
46                                                         self.y: y_validation})
47                #validation_accuracy = self.calculate_accuracy(sess,
48                #                                           X_validation, y_validation)
49                validation_accs.append(validation_accuracy)
50                if best_accuracy < validation_accuracy:
51                    if validation_accuracy / best_accuracy >= \
52                        improve_threthold:
53                        no_improve_steps = 0
54                        best_accuracy = validation_accuracy
55                        saver.save(sess, ckpt_file)
56                print('{0}:{1}# train:{2}, validation:{3}'.format(
57                    epoch, batch_idx, train_accuracy,
58                    validation_accuracy))
59            batch_idx += 1

```

```

60     print('test result:{0}'.format(sess.run(self.accuracy,
61         feed_dict={self.X: X_test, self.y: y_test})))
62     plt.figure(1)
63     plt.subplot(111)
64     plt.plot(eval_times, train_accs, 'b-', label='train accuracy')
65     plt.plot(eval_times, validation_accs, 'r-',
66         label='validation accuracy')
67     plt.title('accuracy trend')
68     plt.legend(loc='lower right')
69     plt.show()

```

第 2、3 行：读入训练样本集输入信号集、训练样本集标签集、验证样本集输入信号集、验证样本集标签集、测试样本集输入信号集、测试样本集标签集。

第 4 行：创建模型。

第 5 行：学习整个训练样本集遍数。

第 6 行：初始化 TensorFlow 模型保存和恢复对象 saver。

第 7 行：每隔 50 次迷你批次学习，计算在验证样本集上的精度。

第 8 行：保存在验证样本集上所取得的最好的验证样本集精度。

第 9 行：定义验证样本集上的精度提高 0.5% 时才显著提高。

第 10 行：记录在验证样本集上精度没有显著提高学习迷你批次的次数。

第 11 行：在验证样本集精度最大没有显著提高的情况下，允许学习迷你批次的数量。

第 12 行：是否终止学习过程。

第 13 行：评估验证样本集上识别精度的次数。

第 14 行：用 eval_times 列表保存评估次数，作为后面绘制识别精度趋势图的横坐标。

第 15 行：用 train_accs 列表保存在训练样本集上每次评估时的识别精度，作为后面图形深色曲线的纵坐标。

第 16 行：用 validation_accs 列表保存在验证样本集上每次评估时的识别精度，作为后面图形浅色曲线的纵坐标。

第 17 行：启动 TensorFlow 会话。

第 18 行：初始化全局参数。

第 19、20 行：如果模式为 TRAIN_MODE_CONTINUE，则读入以前保存的 ckpt 模型文件，初始化模型参数。

第 21 行：循环第 22~59 行操作，对整个训练样本集进行一次学习。

第 22、23 行：如果 is_early_stop 为真，则终止本层循环。

第 24 行：初始化批次数号。

第 25~27 行：循环第 28~59 行操作，对一个迷你批次进行学习。

第 28~30 行：如果验证样本集上识别精度没有显著改善的迷你批次学习次数大于最大允许的验证样本集上识别精度没有显著改善的迷你批次学习次数，则将 is_early_stop 置为真，并退出本层循环。这会直接触发第 22、23 行操作，终止外层循环，学习过程结束。

第 30 行：从训练样本集中取出一个迷你批次的输入信号集 X_mb 和标签集 y_mb。

第 31~33 行：调用 TensorFlow 计算模型输出、代价函数，求出代价函数对参数的导数，

并应用梯度下降算法更新模型参数值。

第 34 行：将验证样本集没有显著改善的迷你批次学习次数加 1。

第 35 行：如果连续进行了指定次数的迷你批次学习，则计算统计信息。

第 36 行：识别精度评估次数加 1。

第 37 行：将识别精度评估次数加入 `eval_times`（图形横坐标）列表中。

第 38~40 行：计算整个训练样本集上的识别精度。

第 39 行：将训练样本集上的识别精度加入训练样本集识别精度列表 `train_accs` 中。

第 40~42 行：计算验证样本集上的识别精度。

第 41、42 行：对训练样本集样本顺序进行重排，随机取出一个迷你批次，在这个迷你批次上计算训练样本集上的识别精度。由于训练样本集可能会比较大，计算量会非常大，所以可以用随机迷你批次的识别精度来近似表示整个训练样本集上的识别精度。

第 43 行：将训练样本集上的识别精度加入到训练样本集识别精度列表 `train_accs` 中。

第 44~46 行：求出验证样本集上的识别精度。

第 47、48 行：对验证样本集样本顺序进行重排，随机取出一个迷你批次，在这个迷你批次上计算验证样本集上的识别精度。由于验证样本集可能会比较大，计算量会非常大，所以可以用随机迷你批次的识别精度来近似表示整个验证样本集上的识别精度。

第 49 行：将验证样本集上的识别精度加入验证样本集识别精度列表 `validation_accs` 中。

第 50 行：如果验证样本集上的最佳识别精度小于当前验证样本集上的识别精度，执行第 51~59 行操作。

第 51~53 行：如果当前验证样本集上的识别精度比之前的最佳识别精度提高 0.5% 以上，则将验证样本集没有显著改善的迷你批次学习次数设为 0。

第 54 行：将验证样本集上最佳识别精度的值设置为当前验证样本集上的识别精度值。

第 55 行：将当前模型参数保存到 `ckpt` 模型文件中。

第 56~58 行：打印训练状态信息。

第 59 行：更新迷你批次样本序号。

第 60、61 行：训练完成后，计算测试样本集上的识别精度，并打印出来。

第 62、63 行：初始化 `matplotlib` 绘图库。

第 64 行：绘制训练样本集上识别精度的变化趋势曲线，用蓝色绘制。

第 65、66 行：绘制验证样本集上识别精度的变化趋势曲线，用红色绘制。

第 67 行：设置图形标题。

第 68 行：在右下角添加图例。

第 69 行：绘制具体图像。

运行结果如图 7.7 所示。

```

0:50# train:0.09690909087657928, validation:0.1019999809265137
0:100# train:0.20280000567436218, validation:0.20100000500679016
0:150# train:0.2614363729953766, validation:0.2572000026702881
0:200# train:0.31279999017715454, validation:0.31619998812675476
0:250# train:0.3895818293094635, validation:0.384799987077713
0:300# train:0.46618181467056274, validation:0.47760000824928284
0:350# train:0.5264000296592712, validation:0.5529999732971191
0:400# train:0.5729818344116211, validation:0.5831999778747559
1:50# train:0.6320909261703491, validation:0.6478000283241272
1:100# train:0.6102181673049927, validation:0.6173999905586243
1:150# train:0.684072732925415, validation:0.6930000185966492
1:200# train:0.7028363347053528, validation:0.7117999792098999
1:250# train:0.7167817950248718, validation:0.7324000000953674
1:300# train:0.7386545538902283, validation:0.756600022315979
1:350# train:0.7574363350868225, validation:0.7666000127792358
1:400# train:0.7654908895492554, validation:0.7753999829292297
2:50# train:0.778109073638916, validation:0.7875999808311462
2:100# train:0.7835999727249146, validation:0.7833999991416931
2:150# train:0.809181809425354, validation:0.8234000205993652
2:200# train:0.8035818338394165, validation:0.8144000172615051
2:250# train:0.8218363523483276, validation:0.8274000287055969
2:300# train:0.8269272446632385, validation:0.8379999995231628
2:350# train:0.8124727010726929, validation:0.8227999806404114
2:400# train:0.84947270154953, validation:0.8547999858856201
3:50# train:0.8461090922355652, validation:0.8474000096321106
3:100# train:0.8423272967338562, validation:0.8461999893188477
3:150# train:0.8518000245094299, validation:0.8568000197410583
test result:0.8644999861717224

```

图 7.7 长短时记忆网络运行后台输出

在训练样本集和验证样本集上识别精度变化趋势曲线如图 7.8 所示。

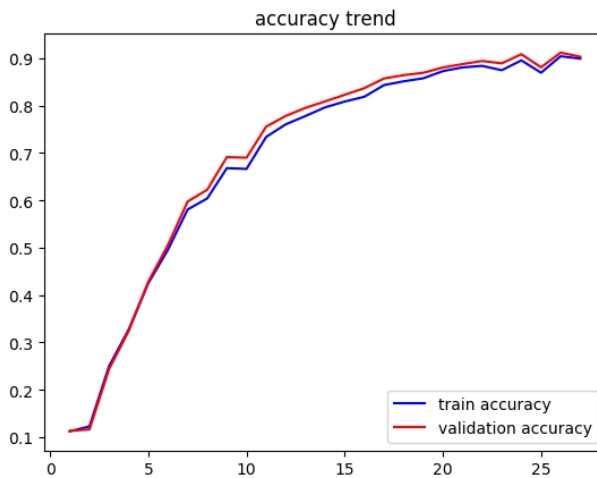


图 7.8 识别精度变化趋势曲线

通过调整各个超参数，采用长短时记忆网络可以达到 99% 以上的精度，但是需要运行的时间也非常长。在这里我们加入了 Early Stopping 调整项，由于停止条件定得比较宽松，所以精度还不是很很高，增加 `max_no_improve_steps = 200` 的值，可以提高识别精度。读者还可以尝试添加 L2 调整项（权值衰减），以提高识别精度；也可以尝试添加 Dropout 调整项，但是在长短时记忆网络中加入 Dropout 调整项通常效果并不太好，需要很多技巧，具体方法可以参考 Google DeepMind 团队的最新论文（<https://arxiv.org/pdf/1409.2329.pdf>）。

由于我们的样本集上的样本比较少，只有 5 万多个，所以上面的代码中求的是整个样本集上的识别精度。如果样本集非常大，包括千万级甚至亿级的样本，在整个样本集上求识别精度运算量就太大了，此时需要用到在随机迷你批次上求识别精度的函数，代码如下：

```

1 def calculate_accuracy(self, sess, X, y):
2     # 计算随机批次上的精度
3     indices = np.arange(len(X)) # Get A Test Batch
4     np.random.shuffle(indices)
5     indices = indices[0:self.batch_size]
6     return np.mean(np.argmax(y[indices], axis=1) ==
7                     np.argmax(sess.run(self.y_,
8                     feed_dict={self.X: X[indices]}), axis=1))

```

第 3 行：求出要求识别精度的样本集的索引号范围，如（0,1,2,3,4,5,...）。

第 4 行：将这些索引号进行洗牌，打乱顺序，例如变为（102,298,301,424,223,...）。

第 5 行：从中取出一个迷你批次大小的索引号。

第 6~8 行：首先在正确分类结果集 y 中按随机索引号取出一个迷你批次的记录，求出每条记录概率值最大的索引号，我们称之为正确索引值。然后在计算出的分类结果集 $y_{_}$ 中按随机索引号取出一个迷你批次的记录，求出每条记录概率值最大的索引号，我们称之为计算索引值。判断正确索引值与计算索引值是否相等，利用 `np.mean` 函数求出正确索引值与计算索引值相等的记录占总体记录的比例。返回该比例值。

如果将识别精度计算替换为在随机迷你批次上的精度，重新训练模型，识别精度变化趋势如图 7.9 所示。

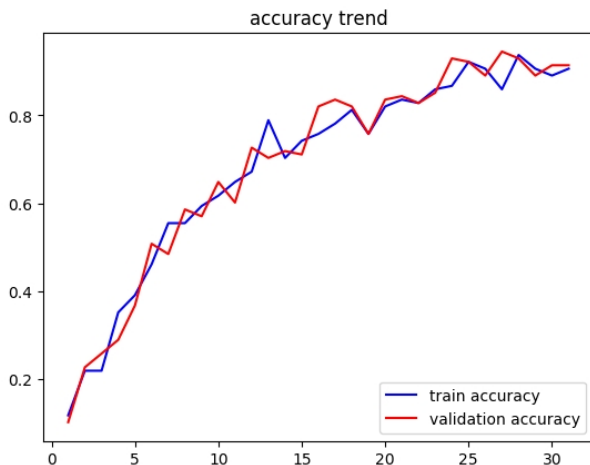


图 7.9 随机迷你批次上识别精度变化趋势

由上图可以看出，识别精度曲线显示出了明显的锯齿状，而整个样本集上的识别精度曲线却比较平滑，这是由随机迷你批次选取时的随机性造成的。但是这两个识别精度变化趋势图的变化趋势是基本一致的，而采用随机迷你批次却可以大幅降低计算量，因此在实际应用中，如果样本集中样本数量很大的话，一般建议采用随机迷你批次上的识别精度作为 Early Stopping 的结束条件。

下面我们来看运行方法，将生成的数字 5 的图片输入到长短时记忆网络中，并将网络的识别结果显示出来，代码如下：

```

1 def run(self, ckpt_file='work/lgr.ckpt'):
2     img_file = 'datasets/test5.png'

```



```

3  img = io.imread(img_file, as_grey=True)
4  raw = [1 if x<0.5 else 0 for x in img.reshape(784)]
5  sample = np.array(raw)
6  self.build_model()
7  X_run = sample.reshape(1, 28, 28)
8  saver = tf.train.Saver()
9  digit = -1
10 with tf.Session() as sess:
11     sess.run(tf.global_variables_initializer())
12     saver.restore(sess, ckpt_file)
13     rst = sess.run(self.y_, feed_dict={self.X: X_run})
14     digit = np.argmax(rst)
15 img_in = sample.reshape(28, 28)
16 plt.figure(1)
17 plt.subplot(111)
18 plt.imshow(img_in, cmap='gray')
19 plt.title('result:{0}'.format(digit))
20 plt.axis('off')
21 plt.show()

```

第 2 行：用绘图软件做一个 28×28 的图像，在上面写一个数字，这里直接写一个印刷体的 5。

第 3 行：以灰度图像方式读出图像内容。

第 4 行：首先将其形状从二维 28×28 变为一维 784，然后根据每个像素值进行处理：值小于 0.5 时取 1，否则取 0。

第 5 行：将其变为 numpy 数组，作为一个样本。

第 6 行：建立模型。

第 7 行：将图片变为长短时记忆网络需要的输入格式。其格式为：（迷你批次大小为 1，时间点为 28 个，输入向量维度为 28）。

第 8 行：利用 tf.train.Saver 函数恢复网络的参数。

第 9 行：记录识别正确的数字。

第 10 行：启动 TensorFlow 会话。

第 11 行：初始化全局参数。

第 12 行：从训练过程中保存的 ckpt 文件中恢复网络参数。

第 13 行：将图片数据输入到网络，计算网络输出层输出 rst，因为此迷你批次大小为 1，所以输出 rst 为一个维度为 10 的向量，其中最大值元素对应的索引值即识别结果。

第 14 行：求出输出 rst 最大值元素所对应的索引值，即识别结果。

第 15 行：将输入信号变为 28×28 的黑白图像格式。

第 16 行：初始化 plt，用于图形绘制。

第 17 行：在 1 行 1 列的第 1 个位置进行绘图。

第 18 行：以灰度图像方式显示输入图像。

第 19 行：将识别结果作为图像的标题进行显示。

第 20 行：不显示坐标轴。

第 21 行：具体显示图像。

运行结果如图 7.10 所示。

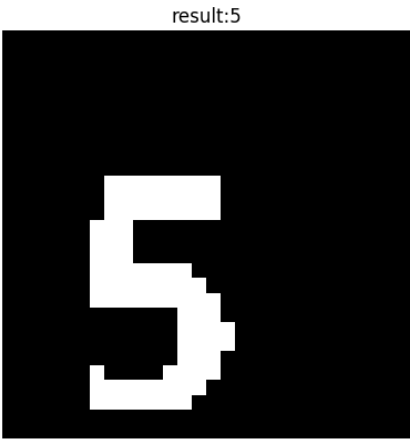


图 7.10 运行结果

由图可知，长短时记忆网络可以正确识别这个图片。需要注意的是，如果读者重新运行一遍训练方法，重新训练出来的网络很可能就不能正确识别这个样本了。这是什么原因呢？因为我们的训练样本是手写数字识别，其中的样本与这里的图片的差别还是很大的，我们的数字不在中心位置，而且非黑即白，而训练样本普遍有灰色的渐变，所以这个样本可以视为对抗性样本，多数网络都会产生错误的识别结果。这与网络模型的选择和训练方法没有太大的关系，而是当前深度学习方法普遍存在的问题。

第三部分 深度学习算法进阶

- ☐ 自动编码器
- ☐ 堆叠自动编码器
- ☐ 受限玻尔兹曼机
- ☐ 深度信念网络

第 8 章

自动编码器

截至目前为止，我们所讨论的神经网络技术，如线性回归模型、逻辑回归模型、多层感知器（MLP）、多层卷积神经网络（CNN），都可以视为前馈神经网络的变形，都会采用信号前向传播及误差反向传播修正连接权值，采用有监督学习方式，解决样本分类问题。

从本章开始，我们将介绍与此有些不同的神经网络架构，即自动编码器的网络。自动编码器属于非监督学习，不需要对训练样本进行标记。自动编码器由三层网络组成，输入层神经元数量与输出层神经元数量相等，中间层神经元数量少于输入层和输出层神经元数量。在网络训练期间，每个训练样本经过网络都会在输出层产生一个新的信号，网络学习的目的就是使输出信号与输入信号尽量相似。自动编码器训练结束之后由两部分组成，首先是输入层和中间层，可以用这个网络对信号进行压缩；其次是中间层和输出层，可以将压缩的信号进行还原。

在本章中，我们将先讲述自动编码器原理；然后分别介绍自动编码器的两种主流扩展：去噪自动编码器、稀疏自动编码器；最后分别给出运行实例，讲述一个自动编码器的实际应用场景。自动编码器最主要的用途是采用分层训练，堆叠成深层网络，这种应用场景将在下一章讲述，并以 MNIST 手写数字识别为例讲述堆叠去噪自动编码器的应用。

8.1 自动编码器概述

自动编码器是当前深度学习研究的热点之一，被使用在很多重要的应用领域。这里仅举一个有趣的例子，大家知道百度推出的上传你的照片，系统帮你找到与你像的明星这个活动吗？其实这个功能就可以用自动编码器来实现，首先，我们将已经训练好的自动编码器的输入层和中间层组成的网络拿出来，将所有明星的脸进行压缩，得到一个人脸向量，

将其保存起来。然后，在普通用户上传了自己的照片后，系统将用户照片输入自动编码器的输入层，从中间层得到该用户的人脸向量。最后，用用户的人脸向量与明星的人脸向量进行比较，找出向量距离最近的一个或几个明星，再将明星的照片作为与用户最像的照片返回给用户。百度这项服务推出的时间较早，所以它应该不是基于自动编码器实现的，但是使用自动编码器完全可以实现这个功能，有兴趣的读者可以试一下。

8.1.1 自动编码器原理

假设有一个 d 维输入信号 \mathbf{x} ，经过输入层到达中间层，信号变为 \mathbf{y} ，可以用以下公式表示：

$$\mathbf{y} = f(\mathbf{W}\mathbf{x} + \mathbf{b}) \quad (1)$$

式中， f 为一个非线性函数，如 Sigmoid 函数、双曲正切函数或 ReLU 函数。 \mathbf{W} 是输入层到中间层的连接权值， \mathbf{b} 为中间层的 Bias 值，信号 \mathbf{y} 经过解码层解码，输出到 d 个神经元的输出层，公式如下：

$$\mathbf{z} = f(\mathbf{W}'\mathbf{y} + \mathbf{b}') \quad (2)$$

式中， s 为一个非线性函数，如 Sigmoid 函数。 \mathbf{W}' 是中间层到输出层的连接权值， \mathbf{b}' 为输出层的 Bias 值。

在通常情况下，有 $\mathbf{W}' = \mathbf{W}^T$ ，即中间层到输出层连接权值矩阵是输入层到中间层连接权值矩阵的转置，当然可以没有这种关系，不过多数情况下取二者相等，在本书中，我们就取二者相等。这时整个网络的参数就变为： $\mathbf{W}, \mathbf{b}, \mathbf{b}'$ 。

当前的问题就变成通过调整网络参数 $\mathbf{W}, \mathbf{b}, \mathbf{b}'$ 使得最终输出 \mathbf{z} 与原始输入信号 \mathbf{x} 尽量接近。

如果将输入层与中间层之间的函数 f 变为线性函数，将最终输出层信号 \mathbf{z} 与原始输入信号的误差变为平方误差，则：

$$L(\mathbf{x}, \mathbf{z}) = \|\mathbf{x} - \mathbf{z}\|^2 \quad (3)$$

则这个问题就变成了一个线性代数中的主成分分析问题。假设中间层有 k 个节点，就变成由输入信号 \mathbf{x} 的前 k 个主成分项来近似表示原始输入信号，我们会在深度学习数学基础的线性代数章节中详细分析这一过程。

因为 \mathbf{y} 可以视为 \mathbf{x} 的有损压缩形式，通过优化算法，可以对训练样本产生很好的压缩效果，同时在测试样本集上有很好的表现，但是并不能保证网络在所有样本上都有好的压缩效果。

8.1.2 去噪自动编码器

我们可以通过在编解码过程中添加随机噪声来解决这一问题，这也是我们把现在要介

绍的网络称为去噪自动编码器的原因。

这个方法的核心思想很简单，就是随机地将原始信号的一些维度数值变为 0，有时甚至可以达到一半左右，将这个加入随机噪声的信号输入到去噪自动编码器，计算得到的输出与原始输入信号之间的误差，再采用前面的随机梯度下降算法对权值进行调整，使误差达到最小。如果网络可以做到这一点，就可以视为网络自动将人为加入的随机噪声去掉了。

为什么要这样做呢？以图像信号为例，很多信号内容是冗余的，去掉之后不影响信息量，例如当图像信号有一些雪花时，还是可以识别出图像内容的。这说明让自动编解码机去噪是可能的。加入噪声是因为人对图像内容的理解，完全可以在某些信息缺失的情况下得出结论，如图像中的物体被遮盖或破损时，我们依然可以识别图像中的物体。去噪自动编码器通过加入随机噪声，试图使神经网络也具有与人类类似的能力。

8.1.3 稀疏自动编码器

随着对自动编码器的研究，研究者们发现，当中间层神经元个数大于输入层神经元个数时，采用非线性编码和随机梯度下降算法时，网络误差和泛化的效果会更好一些。

这是什么原因呢？因为采用的是随机梯度下降和早期终止优化算法，自动编码器编码层非线性神经元的连接权值要非常小，这样才能模拟出线性变换的效果。同时，解码网络又需要非常大的权值才能有效恢复原始信号（可以想象成编码层是乘以一个数，而解码层是乘以这个数的倒数），而优化算法很难产生特别大的连接权值。因此，自动编码器只能拟合于训练样本集。增加中间层神经元的数量，可以在一定程度上解决这个问题。

但是，如果把自动编码器的编码层视为对输入信号的压缩，且中间层的神经元数量多于输入层，那么不仅不能实现压缩，反而使信号扩大了。为此，很多研究人员提出，可以通过向中间层神经元引入稀疏性，使大部分神经元为 0 或接近 0，从而达到信号压缩的目的。从实践来看，这种方法产生的效果还不错。

由于在实际中得到了广泛应用，我们将在本节详细介绍稀疏自动编码器理论。

假设我们有一组没有标记的训练样本： $\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(m)}\}$ ， $\mathbf{x}^{(i)} \in \mathbb{R}^n$ 。

我们将这组训练样本输入到自动编码器，自动编码器的输出层输出 $\mathbf{y}^{(i)} \in \mathbb{R}^n$ ，与输入信号的维度相同，自动编码器通过误差反向传播算法，使得输出信号 $\mathbf{y}^{(i)}$ 与输入信号 $\mathbf{x}^{(i)}$ 尽量相同。同时，由于可以自由选择隐藏层神经元数目，所以可以实现各种需要的特性。在这节里，我们主要通过隐藏层的稀疏性来实现输入信号的特征提取。

自动编码器的网络架构如图 8.1 所示。

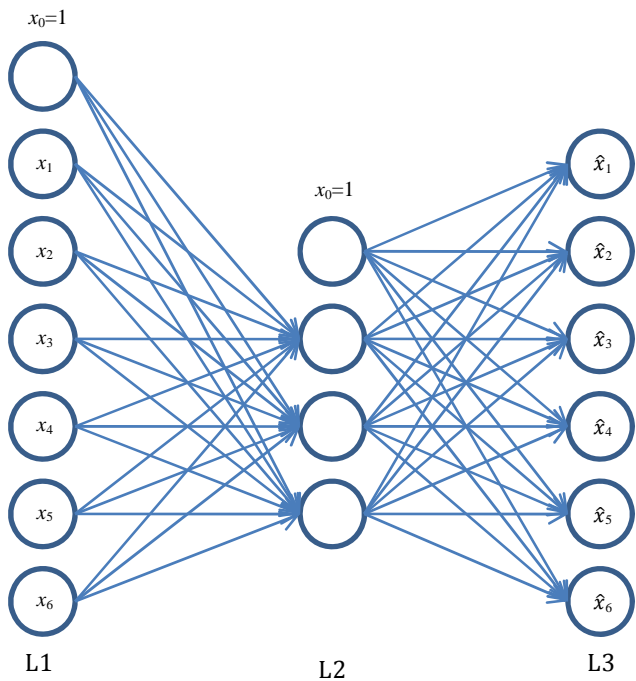


图 8.1 自动编码机的网络架构

以 MNIST 手写数字识别为例，我们知道图像分辨率为 28×28 ，所以输入信号为 $28 \times 28 = 784$ 维，即 $n=784$ 。假设中间层的神经元数为 500，输出层神经元数为 784。先来介绍一下所使用的符号，详细信息见第 4 章多层感知器模型。用 \mathbf{x} 表示输入信号，其上标表示是第几个训练样本；用 \mathbf{a} 表示网络层输出，其上标表示是哪层的输出；用 n_2 表示第二层神经元个数。现在的网络训练任务就是： $\mathbf{x}^{(i)} = \mathbf{a}^{(1)} = \mathbf{a}^{(3)}$ ，即输出信号与输入信号相同。这样训练完成的网络，如果将 $\mathbf{a}^{(2)}$ 输入到输出层，则可以在输出层得到原始的输入信号。由上面的分析可以看出，可以将中间层的输出视为原始输入信号的压缩信号，同时可以视中间层输出为原始输入信号的特征。正是因为以上特性，才使自动编码器在深度学习中得到了越来越广泛的应用。

但是使用自动编码器有一个难点，就是输出层输出 $\mathbf{a}^{(2)}$ 可以视为 \mathbf{x} 的有损压缩形式，通过优化算法，可以对训练样本产生很好的压缩效果，同时在测试样本集上有很好的表现，但是并不能保证网络在所有样本上都有好的压缩效果。

要想解决这个问题，基本有两种策略：一种策略是使用去噪自动编码器，另一种策略就是使用稀疏自动编码器。

稀疏自动编码器的基本思想是不要求中间层神经元数量小于输入层，但是限制中间层必须具有稀疏性，这样虽然中间层神经元数量可能多于输入信号，但是由于具有稀疏性，也可以压缩输入信号或发现原始输入信号的特征。

我们设定中间层神经元采用 Sigmoid 函数，规定当神经元输出接近 1 的时候，称其为激活状态；当神经元输出接近 0 时，称其为非激活状态。由于稀疏性可知，中间层神经元大部分时间都应该处于非激活状态。而且在某一时刻，中间层多数神经元也需要处于非激活状态。

在多层感知器模型中，采用 $a_j^{(i)}$ 代表第 i 层第 j 个神经元的输出，这里我们仍沿用这个表示法，但是稀疏自动编码器具有稀疏性，所以中间层神经元在多数时间都处于非激活状态，只有在少数情况下才处于激活状态。我们希望记录下中间层神经元在每个输入信号的状态，因此引入以下符号： $a_j^{(2)}(\mathbf{x}^{(i)})$ ，代表第2层第 j 个神经元在第 i 个样本下的输出值。

定义中间层神经元在整个训练样本集上的平均激活值为：

$$\hat{\rho}_j = \frac{1}{m} \sum_{i=1}^m [a_j^{(2)}(\mathbf{x}^{(i)})] \quad (4)$$

定义稀疏参数为 ρ ，其为一个接近0的数，并且希望 $\rho = \hat{\rho}_j$ 。在实际中，稀疏参数通常取0.05。

我们希望 $\hat{\rho}_j = \rho$ ，可以视为一个优化问题，即希望中间层每个神经元的激活都与稀疏参数尽量接近。为此可以对与稀疏参数不一致的中间层神经元进行惩罚，例如：

$$\sum_{j=1}^{n_2} \left(\rho \log \frac{\rho}{\hat{\rho}_j} + (1 - \rho) \log \frac{1 - \rho}{1 - \hat{\rho}_j} \right) \quad (5)$$

其实上式就是 KL 散度公式，用于衡量 $\hat{\rho}_j$ 、 ρ 之间的相似程度，KL 散度的定义为：

$$\sum_{j=1}^{n_2} \text{KL}(\rho || \hat{\rho}_j) \quad (6)$$

我们知道， $\text{KL}(\rho || \hat{\rho}_j)$ 表示以 ρ 为均值的伯努利分布变量和以 $\hat{\rho}_j$ 为均值的伯努利分布变量之间的相似关系。如果 $\rho = \hat{\rho}_j$ ，则 $\text{KL}(\rho || \hat{\rho}_j) = 0$ ， ρ 、 $\hat{\rho}_j$ 相差越大， $\text{KL}(\rho || \hat{\rho}_j)$ 的值也越大。

对于稀疏自动编码器我们可以定义以下代价函数：

$$J_{\text{sparse}}(\mathbf{W}, \mathbf{b}) = J(\mathbf{W}, \mathbf{b}) + \beta \sum_{j=1}^{n_2} \text{KL}(\rho || \hat{\rho}_j) \quad (7)$$

式中， $J(\mathbf{W}, \mathbf{b})$ 为在多层感知器中定义的代价函数， β 为稀疏惩罚项系数。因为 \mathbf{W} 、 \mathbf{b} 决定中间层神经元输出，因此可以决定 $\hat{\rho}_j$ 。

稀疏自动编码器的训练过程与多层感知器模型非常相似，所以在这里就不详细展开了，只讲解其中不同的部分，其余部分读者可以参考多层感知器模型的介绍。

我们来回忆一下多层感知器模型训练算法，首先需要计算各层误差 $\delta_j^{(i)}$ ，然后求代价函数对参数的偏导，最后调整参数值。在这里，只是求中间层误差时与多层感知器模型不同，

其余步骤是完全相同的。而中间层误差计算公式为：

$$\delta_i^{(2)} = \left(\left(\sum_{j=1}^{n_3} w_{ji}^{(2)} \delta_j^{(3)} \right) + \beta \left(-\frac{\rho}{\hat{\rho}_j} + \frac{1-\rho}{1-\hat{\rho}_j} \right) \right) \quad (8)$$

但是要计算上式中的误差，需要先计算出 $\hat{\rho}_j$ 的值，而计算 $\hat{\rho}_j$ 就必须将所有训练样本进行一次前向传播才能得到，如果整个训练样本集不是很大还比较好处理，但是如果整个训练样本集非常大，不能一次性装入内存进行计算，那么就需要采用特殊的方法进行计算了。我们需要先遍历整个训练样本集，仅求出中间层输出即可计算出所有 $\hat{\rho}_j$ 的值并记录下来。然后我们再按正常方式，运行训练样本集中的每个样本，先正向传播再求出误差，此时 $\hat{\rho}_j$ 取前面已经计算好的值，再进行权值调整。在进行下一个样本前，需要重复上面的步骤。由此可见，当训练样本集非常大时，稀疏自动编码器需要对每个训练样本进行两次前向传播的计算。

8.2 去噪自动编码器 TensorFlow 实现

我们知道去噪自动编码器的工作主要由四步组成：第一步是向原始输入信号中加入随机噪声（使原始信号在某些维度上值为零）；第二步是将加入噪声的信号输入网络，经过编码器部分，在中间层生成输入信号的压缩信号；第三步是经过解码机层，在输出层得到输出信号；第四步是将输出信号与原始输入信号相比较，求出误差，并根据随机梯度下降算法更新网络的连接权值。我们先来看载入 MNIST 手写数字识别模型程序，代码如下：

```
1 def load_datasets(self):
2     mnist = input_data.read_data_sets(self.datasets_dir,
3         one_hot=True)
4     X_train = mnist.train.images
5     y_train = mnist.train.labels
6     X_validation = mnist.validation.images
7     y_validation = mnist.validation.labels
8     X_test = mnist.test.images
9     y_test = mnist.test.labels
10    return X_train, y_train, X_validation, y_validation, \
11        X_test, y_test, mnist
```

第 2、3 行：调用 TensorFlow 的 input_data 的 read_data_sets 方法，第一个参数为数据集存放路径，第二个参数是标签集的格式。在原始 MNIST 数据集中，我们知道每个样本是 28×28 的黑白图片，对应的是 0~9 的数字标签，所以其格式为：[...784（28×28）像素点的值...][3]。其中，第一项为 784（28×28）个 0~1 的浮点数，0 代表黑色，1 代表白色；第二项的“3”代表这个样本是数字 3。为了后续处理方便，我们将标签[3]改为 one-hot 形式，因为标签代表 0~9 的数字，所以标签集为 10 维向量，每维上取值为 0 代表不是这个对应

位置的数字，取值为 1 代表是这个对应位置的数字。其中，只有一维可以取 1，因此称之为 one-hot，还以上面的例子为例，标签集的格式就变为：[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]，因为第四位为 1，所以代表这个样本是数字 3。

第 4 行：取出训练样本集输入信号集 X_{train} ，其为设计矩阵形式，每一行代表一个样本，行数为训练样本集中的样本数量。在这个例子中，就 $X_{\text{train}} \in \mathbb{R}^{55000 \times 784}$ ，其中训练样本集中有 55000 个样本，每个样本是 784 (28×28) 维的图片。

第 5 行：取出训练样本集标签集 y_{train} ，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为训练样本集中的样本数量。在这个例子中，就 $y_{\text{train}} \in \mathbb{R}^{55000 \times 10}$ ，其中训练样本集中有 55000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 6 行：取出验证样本集输入信号集 $X_{\text{validation}}$ ，其为设计矩阵形式，每一行代表一个样本，行数为验证样本集中的样本数量。在这个例子中，就 $X_{\text{validation}} \in \mathbb{R}^{5000 \times 784}$ ，其中验证样本集中有 5000 个样本，每个样本是 784 (28×28) 维的图片。根据前面我们的讨论可以知道，在训练过程中，为了防止模型出现过拟合，模型的泛化能力降低（模型在训练样本集达到非常高的精度，但是在未见过的测试样本集或实际应用中，精度反而不高），通常会采用 Early Stopping 策略，就是在逻辑回归模型训练过程中，只用训练样本集对模型进行训练，每隔一定的时间间隔，计算模型在未见过的验证样本集上识别的精度，并记录迄今为止在验证样本集上取得的最高精度。我们会发现，在训练初期，验证样本集上的识别精度会稳步提高，但是到了一定阶段之后，验证样本集上的识别精度就不会再明显提高了，甚至开始逐渐下降，这就说明模型出现了过拟合，这时就可以停止模型训练，将在验证样本集上取得最佳识别精度的模型参数作为模型最终的参数。综上所述，验证样本集主要用于防止模型出现过拟合，为 Early Stopping 算法提供终止依据。

第 7 行：取出验证样本集标签集 $y_{\text{validation}}$ ，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为训练样本集中的样本数量。在这个例子中，就 $y_{\text{validation}} \in \mathbb{R}^{5000 \times 10}$ ，其中验证样本集中有 5000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 8 行：取出测试样本集输入信号集 X_{test} ，其为设计矩阵形式，每一行代表一个样本，行数为训练样本集中的样本数量。在这个例子中，就 $X_{\text{test}} \in \mathbb{R}^{10000 \times 784}$ ，其中训练样本集中有 10000 个样本，每个样本是 784 (28×28) 维的图片。测试样本集主要用于模型训练结束后对模型性能进行评估。由于模型没有见过测试样本集中的样本，可以模拟模型在实际部署之后的情况，模型在测试样本集上的识别精度，基本可以视为模型在实际应用中可以达到的精度。

第 9 行：取出测试样本集标签集 y_{test} ，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为测试样本集中的样本数量。在这个例子中，就 $y_{\text{test}} \in \mathbb{R}^{10000 \times 10}$ ，其中测试样本集中有 10000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 10、11 行：返回训练样本集输入信号集、训练样本集标签集、验证样本集输入信号集、验证样本集标签集、测试样本集输入信号集、测试样本集标签集。

下面我们来看去噪自动编码器模型的创建，这个模型输入层有 784 (28×28) 个神经元，隐藏层有 1024 个神经元，输出层有 784 (28×28) 个神经元，所有神经元采用双曲正切函数作为激活函数，代码如下：

```
1 def build_model(self):
2     print('Build Denoising Autoencoder Model v0.0.8')
3     print('begin to build the model')
4     self.X = tf.placeholder(shape=[None, self.n], dtype=tf.float32)
5     self.y = tf.placeholder(shape=[None, self.n], dtype=tf.float32)
6     self.keep_prob = tf.placeholder(dtype=tf.float32, name='keep_prob')
7     self.W1 = tf.Variable(
8         tf.truncated_normal(
9             shape=[self.n, self.hidden_size], mean=0.0, stddev=0.1),
10        name='W1')
11    self.b2 = tf.Variable(tf.constant(
12        0.001, shape=[self.hidden_size]), name='b2')
13    self.b3 = tf.Variable(tf.constant(
14        0.001, shape=[self.n]), name='b3')
15    with tf.name_scope('encoder'):
16        z2 = tf.matmul(self.X, self.W1) + self.b2
17        self.a2 = tf.nn.tanh(z2)
18    with tf.name_scope('decoder'):
19        z3 = tf.matmul(self.a2, tf.transpose(self.W1)) + self.b3
20        a3 = tf.nn.tanh(z3)
21    self.y_ = a3
22    r_y_ = tf.clip_by_value(self.y_, 1e-10, float('inf'))
23    r_l_y_ = tf.clip_by_value(1 - self.y_, 1e-10, float('inf'))
24    cost = - tf.reduce_mean(tf.add(
25        tf.multiply(self.y, tf.log(r_y_)),
26        tf.multiply(tf.subtract(1.0, self.y), tf.log(r_l_y_))))
27    self.J = cost + self.regcoef * tf.nn.l2_loss([self.W1])
28    self.train_op = tf.train.AdamOptimizer(0.001, 0.9, 0.9, 1e-08).minimize(self.J)
```

第 4 行：定义网络输入信号的 placeholder，其为 784 维。在训练过程中，输入到网络的为加入噪声的图像；在运行时，输入网络的则为原始图像。

第 5 行：定义网络输出层输出信号，其为 784 维，我们的目标就是使 y 和 X 尽量相等。

第 6 行：定义采用 Dropout 调整项时，神经元的活跃概率。

第 7~10 行：定义输入层到隐藏层的连接权值对象，维度为 784×1024，用均值为 0、标准差为 0.1 的正态分布随机数进行初始化。

第 11、12 行：定义隐藏层偏置值，采用均值为 0、标准差为 0.001 的正态分布随机数进行初始化。

第 13、14 行：定义输出层偏置值，采用均值为 0、标准差为 0.001 的正态分布随机数进行初始化。注意：由于隐藏层到输出层的连接权值矩阵是输出层到隐藏层连接权值矩阵的转换，所以不需要定义隐藏层到输出层的连接权值矩阵。

第 15~17 行：定义输入层到隐藏层的编码器部分，首先将输入信号与连接权值矩阵相乘，再加上隐藏层偏置值，最后经过双曲正切激活函数，求出隐藏层输出。读者可以参照

前面章节的内容加入 Dropout 调整项，再求出输出。这里我们没有给出代码，读者可以将其当做一个练习，加入适当的 Dropout，尝试重新训练网络。

第 18~21 行：定义隐藏层到输出层的解码机部分，将隐藏层输出信号与输入层到隐藏层连接权值矩阵 $W1$ 的转置相乘，加上输出层的偏置值，经过双曲正切激活函数，求出输出层的输出信号 y_- 。

第 22、23 行：因为用计算机表示浮点数有精度问题，所以对 y_- 和 $1-y_-$ ，如果值小于 $1e-10$ ，则取值为 $1e-10$ 。

第 24~26 行：定义代价函数，假设有 m 个训练样本，则计算公式为：

$$\text{cost} = y \log \hat{y} + (1 - y) \log(1 - \hat{y})$$

第 27 行：定义最终的代价函数为加上 L2 调整项（权值衰减）后的值，其中 `tf.nn.l2_loss` 函数用于计算权值衰减项，这里只对连接权值进行调整。`self.regcoef` 为连接权值衰减项的系数。

第 28 行：定义训练操作，采用 Adam 优化算法，求代价函数为最小值时参数的值。

定义完网络模型之后，就可以进行网络训练了，代码如下：

```
1 def train(self, mode=TRAIN_MODE_NEW, ckpt_file='work/dae.ckpt'):
2     X_train, y_train, X_validation, y_validation, \
3         X_test, y_test, mnist = self.load_datasets()
4     with self.tf_graph.as_default():
5         self.build_model()
6         saver = tf.train.Saver()
7         with tf.Session() as sess:
8             sess.run(tf.global_variables_initializer())
9             for epoch in range(self.num_epochs):
10                 X_train_prime = self.add_noise(sess, X_train,
11                                                 self.corr_frac)
12                 shuff = list(zip(X_train, X_train_prime))
13                 np.random.shuffle(shuff)
14                 batches = [_ for _ in self.gen_mini_batches(shuff,
15                                                             self.batch_size)]
16                 batch_idx = 1
17                 for batch in batches:
18                     X_batch_raw, X_prime_batch_raw = zip(*batch)
19                     X_batch = np.array(X_batch_raw).astype(np.float32)
20                     X_prime_batch = np.array(X_prime_batch_raw).\
21                         astype(np.float32)
22                     batch_idx += 1
23                     opv, loss = sess.run([self.train_op, self.J], feed_dict={
24                                             self.X: X_prime_batch, self.y: X_batch})
25                     if batch_idx % 100 == 0:
26                         print('epoch{0}_batch{1}: {2}'.format(epoch,
27                                                                 batch_idx, loss))
28                 saver.save(sess, ckpt_file)
```

第 2、3 行：载入 MNIST 数据集，得到训练样本集输入样本集、训练样本集标签集、验证样本集输入样本集、验证样本集标签、测试样本集输入样本集、测试样本集标签集。

第 4 行：启动 TensorFlow 的 graph。

第 5 行：调用 `build_model` 创建去噪自动编码器模型。

第 6 行：定义 `tf.train.Saver` 对象，用于保存训练阶段的参数值等信息。

第 7 行：启动 `TensorFlow` 会话。

第 8 行：初始化全局变量。

第 9 行：对每个全量训练样本集训练，执行第 10~28 行操作。

第 10、11 行：向训练样本集输入样本集添加 `self.corr_frac` 比例的 `Masking` 噪声。这就是这个模型被称为去噪自动编码器模型的原因，网络的任务除了恢复输入信号，还需要去掉人为加上去的噪声。

第 12 行：把原始训练样本集输入样本集和加入噪声的训练样本集输入样本集，以样本一一对应的方式形成一个列表。

第 13 行：对上一步操作形成的列表进行随机洗牌。

第 14、15 行：以 `batch_size` 为单位，将样本集拆分为迷你批次。

第 16 行：记录迷你批次序号。

第 17 行：对于每个迷你批次，循环第 18~28 行操作。

第 18 行：从批次中取出原始样本集和加入噪声的样本集。

第 19 行：将原始样本集变为 `numpy.ndarray` 类型。

第 20、21 行：将加入噪声的样本集变为 `numpy.ndarray` 类型。

第 22 行：求出迷你批次序号。

第 23、24 行：通过 `TensorFlow` 求出代价函数值。注意，此时 `self.X` 的输入为加入噪声的样本集，`self.y` 输入的是原始样本集。

第 25~27 行：每隔 100 次打印一次训练信息。

第 28 行：保存网络参数。

由于我们的网络比较小，所以运行训练方法时只需要几分钟就可以完成训练过程。

在训练过程中，还用到了两个方法：向输入样本集加入 `Masking` 噪声和产生迷你批次。

下面我们分别来看这两个函数的实现。

向输入样本集加入 `Masking` 噪声的代码如下：

```
1 def add_noise(self, sess, X, corr_frac):
2     X_prime = X.copy()
3     rand = tf.random_uniform(X.shape)
4     X_prime[sess.run(tf.nn.relu(tf.sign(corr_frac - rand))).astype(np.bool)] = 0
5     return X_prime
```

第 2 行：复制一份输入样本集。

第 3 行：利用均匀分布产生 0~1 的随机数。

第 4 行：将产生的随机数与噪声比例相减，取其符号，经过 `ReLU` 函数，如果为负数时值为 0，如果为正数其值不变，也就是说，有 `corr_frac` 比例的输入信号将被置 0。

第 5 行：返回加入噪声的样本集。

产生迷你批次的代码如下：

```
1 def gen_mini_batches(self, X, batch_size):
2     X = np.array(X)
3     for i in range(0, X.shape[0], batch_size):
4         yield X[i:i + batch_size]
```

这段代码先将样本变为 `numpy` 的数组，将其分割为 `batch_size` 大小的子数组，再用 `yield` 函数将其作为一个序列。

运行训练方法会产生类似图 8.2 的输出结果。

```
epoch9_batch3400: 0.365437775850296
epoch9_batch3500: 0.225712850689888
epoch9_batch3600: 0.2144874930381775
epoch9_batch3700: 0.4417518675327301
epoch9_batch3800: 0.0050604743883013725
epoch9_batch3900: 0.41609320044517517
epoch9_batch4000: 0.11224576830863953
epoch9_batch4100: 0.17788507044315338
epoch9_batch4200: 0.27541428804397583
epoch9_batch4300: 0.18187116086483002
epoch9_batch4400: -0.02158653736114502
epoch9_batch4500: 0.2431483417749405
epoch9_batch4600: 0.03714064508676529
epoch9_batch4700: 0.32946839928627014
epoch9_batch4800: 0.13775542378425598
epoch9_batch4900: 0.09773144125938416
epoch9_batch5000: 0.2855677902698517
epoch9_batch5100: 0.014772463589906693
epoch9_batch5200: 0.155035138130188
epoch9_batch5300: 0.4509543478488922
epoch9_batch5400: 0.3788836598396301
epoch9_batch5500: 0.3795548975467682
```

图 8.2 去噪自动编码器训练过程输出结果

在网络训练完成之后，我们就需要将网络置于运行状态，这时输入网络的是原始输入信号，要得到的是隐藏层的表示结果，代码如下：

```
1 def run(self, ckpt_file='work/dae.ckpt'):
2     img_file = 'datasets/test5.png'
3     img = io.imread(img_file, as_grey=True)
4     raw = [1 if x<0.5 else 0 for x in img.reshape(784)]
5     sample = np.array(raw)
6     X_run = sample.reshape(1, 784)
7     digit = -1
8     with self.tf_graph.as_default():
9         self.build_model()
10        saver = tf.train.Saver()
11        with tf.Session() as sess:
12            sess.run(tf.global_variables_initializer())
13            saver.restore(sess, ckpt_file)
14            hidden_data, output_data = sess.run([self.a2, self.y_],
15                                                feed_dict={self.X: X_run})
16        img_in = sample.reshape(28, 28)
17        plt.figure(1)
18        plt.subplot(131)
19        plt.imshow(img_in, cmap='gray')
20        plt.title('origin')
21        plt.axis('off')
22        #
23        plt.subplot(132)
24        hidden_pic = hidden_data.reshape(32, 32)
25        plt.imshow(hidden_pic, cmap='gray')
26        plt.axis('off')
27        plt.title('hidden layer')
28        #
29        plt.subplot(133)
```

```

30     restore_img = output_data.reshape(28, 28)
31     plt.imshow(restore_img, cmap='gray')
32     plt.axis('off')
33     plt.title('restore image')
34     plt.show()

```

第 2 行：指定用于运行的图片文件，这里使用的是一个 28×28 的二维黑白图像数字 5。

第 3 行：读出图像文件中的内容。

第 4 行：小于 0.5 的数值取 0，大于 0.5 的数值取 1。

第 5 行：将其转换为 numpy 数组。

第 6 行：将其变为迷你批次序列，这个序列中只有 1 个迷你批次，且迷你批次中只包括一个样本。

第 7 行：记录样本所代表的数据。

第 8 行：启动 TensorFlow 计算图。

第 9 行：建立去噪自动编码器模型。

第 10 行：初始化 tf.train.Saver 对象。

第 12 行：初始化全局变量。

第 13 行：恢复训练好的模型数据。

第 14、15 行：利用 TensorFlow 的 Session 对象，求出隐藏层的数据和输出层的数据。

第 16 行：将输入图像数据变为 28×28 的二维数组形式。

第 17 行：初始化绘图软件 matplotlib。

第 18~21 行：在 1 行 3 列的网格的第 1 行第 1 列，以黑白图片方式绘制原始图片。

第 23~27 行：在 1 行 3 列的网格的第 1 行第 2 列，以黑白图片方式绘制隐藏层中提取到的图片特征信息。

第 29~33 行：在 1 行 3 列的网格的第 1 行第 3 列，以黑白图片方式绘制恢复出的图片信息。

运行结果如图 8.3 所示。

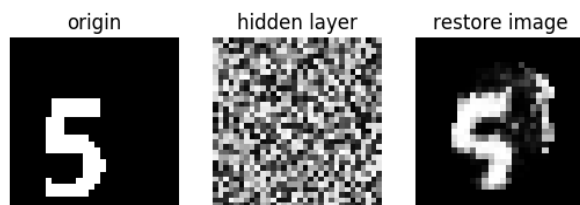


图 8.3 图像编解码举例

从恢复的图片看，虽然图片很不清晰，但是我们还是能大致看出原始图像的影子，这是由于我训练的 epoch 较少造成的，增加 epoch 数可以提高图像复原的质量。自动编码器的隐藏层可以视为降维（Dimension Reduction）或特征工程，即隐藏层可以作为图像的特征更好地代表图像信息，将这些图像特征输入到分类器等网络中，可以取得更好的分类结果。但是，在深度学习特征工程中提取的特征信息，绝大多数是让人无法理解的，就像图 8.3 中间的图片一样，不具有可解释性，这也是深度学习特征工程的一大弊病。

8.3 去噪自动编码器的 Theano 实现

我们知道去噪自动编码器的工作主要由四步组成，通过定义去噪自动编码器（Denosing Autoencoder）类，就可以实现各步的功能，代码如下：

```

1 from __future__ import print_function
2 import os
3 import sys
4 import timeit
5 import numpy
6 import theano
7 import theano.tensor as T
8 from theano.tensor.shared_randomstreams import RandomStreams
9
10 class DenosingAutoencoder(object):
11     def __init__(
12         self,
13         numpy_rng,
14         theano_rng=None,
15         input=None,
16         n_visible=784,
17         n_hidden=500,
18         W=None,
19         bhid=None,
20         bvis=None
21     ):
22         self.n_visible = n_visible
23         self.n_hidden = n_hidden
24         if not theano_rng:
25             theano_rng = RandomStreams(numpy_rng.randint(2 ** 30))
26         if not W:
27             initial_W = numpy.asarray(
28                 numpy_rng.uniform(
29                     low=-4 * numpy.sqrt(6. / (n_hidden + n_visible)),
30                     high=4 * numpy.sqrt(6. / (n_hidden + n_visible)),
31                     size=(n_visible, n_hidden)
32                 ),
33                 dtype=theano.config.floatX
34             )
35             W = theano.shared(value=initial_W, name='W', borrow=True)
36
37         if not bvis:
38             bvis = theano.shared(
39                 value=numpy.zeros(
40                     n_visible,
41                     dtype=theano.config.floatX
42                 ),
43                 borrow=True
44             )
45
46         if not bhid:
47             bhid = theano.shared(
48                 value=numpy.zeros(
49                     n_hidden,
50                     dtype=theano.config.floatX
51                 ),
52                 name='b',
53                 borrow=True
54             )
55         self.W = W
56         self.b = bhid
57         self.b_prime = bvis
58         self.W_prime = self.W.T
59         self.theano_rng = theano_rng
60         if input is None:
61             self.x = T.dmatrix(name='input')
62         else:
63             self.x = input
64         self.params = [self.W, self.b, self.b_prime]
65

```

```

66     def get_corrupted_input(self, input, corruption_level):
67         return self.theano_rng.binomial(size=input.shape, n=1,
68                                         p=1 - corruption_level,
69                                         dtype=theano.config.floatX) * input
70
71     def get_hidden_values(self, input):
72         return T.nnet.sigmoid(T.dot(input, self.W) + self.b)
73
74     def get_reconstructed_input(self, hidden):
75         return T.nnet.sigmoid(T.dot(hidden, self.W_prime) + self.b_prime)
76
77     def get_cost_updates(self, corruption_level, learning_rate):
78         tilde_x = self.get_corrupted_input(self.x, corruption_level)
79         y = self.get_hidden_values(tilde_x)
80         z = self.get_reconstructed_input(y)
81         L = - T.sum(self.x * T.log(z) + (1 - self.x) * T.log(1 - z), axis=1)
82         cost = T.mean(L)
83         gparams = T.grad(cost, self.params)
84         updates = [
85             (param, param - learning_rate * gparam)
86             for param, gparam in zip(self.params, gparams)
87         ]
88
89     return (cost, updates)

```

第 10 行：定义 DenosingAutoencoder 类。

第 11 行：定义 DenosingAutoencoder 类构造函数。

- ☐ numpy_rng: 随机数生成引擎，用于初始化权值。
- ☐ theano_rng: Theano 的随机数生成器。
- ☐ input: 输入信号，采用单独去噪自动编码器时其为空；采用堆叠自动编码器时其为 Theano 变量，由下层计算结果作为输入。
- ☐ n_visible: 输入层神经元个数。
- ☐ W: 连接权值。
- ☐ bhid: 偏移值 Bias，采用单独去噪自动编码器时可忽略。
- ☐ bvis: 偏移值 Bias，采用单独去噪自动编码器时可忽略。

第 22 行：设置可见层神经元数量。

第 23 行：设置隐藏层神经元数量。

第 24、25 行：如果没有指定 Theano 随机数生成引擎，利用 theano_rng 重新生成一个。

第 26~35 行：如果没有指定连接权值矩阵，则用随机数进行初始化，并将其定义为 Theano 的共享变量 W。

第 37~44 行：如果没有指定可见层偏移量 bvis，则用 0 初始化。

第 46~54 行：如果没有指定隐藏层偏移量 bhid，则用 0 初始化。

第 55 行：设置类属性 W 为之前定义的共享变量连接权值 W。

第 56 行：设置类属性 b 为隐藏层偏移量 bhid。

第 57 行：设置类属性 b_prime 为可见层偏移量 bvis。

第 58 行：设置类属性 w_prime 为隐藏层到输出层连接权值，其值为输入层到隐藏层连接权值的转置。

第 59 行：随机数生成引擎，设置类属性 theano_rng。

第 60~63 行：设置类属性 x 为输入信号，如果未指定则初始化一个矩阵。

第 64 行：设置模型参数为连接权值、隐藏层偏移量和可见层偏移量。

第 66~69 行：定义获取加入噪声后输入信号的方法，调用 Theano 方法向输入信号中加入随机噪声。

第 71、72 行：获取隐藏层输入值。

第 74、75 行：获取输出层输出值。

第 77 行：定义计算代价函数和更新参数的方法。

第 78 行：获取加入随机噪声的输入信号。

第 79 行：获取隐藏层的输出信号。

第 80 行：获取输出层的重建信号。

第 81 行：计算总的代价函数值，公式为： $x \cdot \log \hat{x} + (1 - x) \log(1 - \hat{x})$ 。

第 82 行：计算代价函数的平均值。

第 83 行：将代价函数对所有参数求导。

第 84~87 行：定义参数更新规则为 $w = w - \alpha \frac{\partial \mathcal{L}}{\partial w}$ 。

下面来看去噪自动编码器引擎类，代码如下：

```
1 from __future__ import print_function
2 import os
3 import sys
4 import timeit
5 import numpy
6 import theano
7 import theano.tensor as T
8 from theano.tensor.shared_randomstreams import RandomStreams
9 from mnist_loader import MnistLoader
10 from denosing_autoencoder import DenosingAutoencoder
11 from dlt_utils import tile_raster_images
12 try:
13     import PIL.Image as Image
14 except ImportError:
15     import Image
16
17 class DAMnistEngine(object):
18     def train(self, learning_rate=0.1, training_epochs=15,
19             dataset='mnist.pkl.gz',
20             batch_size=20, output_folder='dA_plots'):
21         loader = MnistLoader()
22         datasets = loader.load_data(dataset)
23         train_set_x, train_set_y = datasets[0]
24         n_train_batches = train_set_x.get_value(borrow=True).shape[0]
25         index = T.lscalar()
26         x = T.matrix('x')
27         if not os.path.isdir(output_folder):
28             os.makedirs(output_folder)
29         os.chdir(output_folder)
30         self.train_da(x, learning_rate, train_set_x, index, batch_size, \
31             training_epochs, n_train_batches, \
32             'filters_corruption_0.png', 0.0)
33         self.train_da(x, learning_rate, train_set_x, index, batch_size, \
34             training_epochs, n_train_batches, \
35             'filters_corruption_30.png', 0.3)
36         os.chdir('../')
37
38     def train_da(self, x, learning_rate, train_set_x, index, batch_size,
39             training_epochs, n_train_batches,
40             img_file,
41             corruption_level=0.0):
42         rng = numpy.random.RandomState(123)
43         theano_rng = RandomStreams(rng.randint(2 ** 30))
44         da = DenosingAutoencoder(
45             numpy_rng=rng,
46             theano_rng=theano_rng,
47             input=x,
```

```

48         n_visible=28 * 28,
49         n_hidden=500
50     )
51     cost, updates = da.get_cost_updates(
52         corruption_level=corruption_level,
53         learning_rate=learning_rate
54     )
55     da_net = theano.function(
56         [index],
57         cost,
58         updates=updates,
59         givens={
60             x: train_set_x[index * batch_size: (index + 1) * batch_size]
61         }
62     )
63     start_time = timeit.default_timer()
64     for epoch in range(training_epochs):
65         c = []
66         for batch_index in range(n_train_batches):
67             c.append(da_net(batch_index))
68         print('Training epoch %d, cost ' % epoch, numpy.mean(c))
69     end_time = timeit.default_timer()
70     training_time = (end_time - start_time)
71     print(('The no corruption code for file ' +
72           os.path.split(__file__)[1] +
73           ' ran for %.2fm' % ((training_time) / 60.)), file=sys.stderr)
74     image = Image.fromarray(
75         tile_raster_images(X=da.W.get_value(borrow=True).T,
76                           img_shape=(28, 28), tile_shape=(10, 10),
77                           tile_spacing=(1, 1)))
78     image.save(img_file)
79
80 def run():
81     print('run Denosing Autoencoder...')

```

第 12~15 行：检测系统是否安装 PIL，如果安装则引入 PIL 中的 Imaging，如果没有安装则直接引用 Imaging。

第 17 行：定义 DAMnistEngine 类。

第 18 行：定义网络训练接口方法。

- ☐ learning_rate: 学习率。
- ☐ training_epoch: 训练样本集循环训练最大次数。
- ☐ dataset: MNIST 数据集文件名称。
- ☐ batch_size: 迷你批次大小。
- ☐ output_folder: 将训练结果绘制成图像。

第 21 行：初始化 MNIST 数据载入类。

第 22 行：载入 MNIST 数据集数据。

第 23 行：取出训练样本集数据，将图像数据赋给 train_set_x，将结果标签数据赋给 train_set_y。

第 24 行：求出迷你批次数量。

第 25 行：定义 index 变量保存迷你批次的序列号。

第 26 行：定义变量 x 为矩阵类型，保存输入信号。

第 27、28 行：判断图像输出目录是否存在，如果不存在则创建该目录。

第 29~32 行：调用本类 train_da 方法，随机噪声率为 0。

第 33~35 行：调用本类 train_da 方法，随机噪声率为 30%。

第 36 行：返回上一级目录。

第 38 行：定义 `train_da` 方法。

- ❑ `learning_rate`：参数调整时的学习率。
- ❑ `train_set_x`：训练样本集输入信号。
- ❑ `index`：迷你批次序号。
- ❑ `batch_size`：迷你批次大小。
- ❑ `training_epoch`：训练样本集全部训练的最大次数。
- ❑ `n_train_batches`：训练样本集迷你批次数量。
- ❑ `img_file`：输出图像文件名称。

第 42 行：生成随机数引擎。

第 43 行：生成 Theano 随机数引擎。

第 44~50 行：生成 `DenosingAutoencoder` 类实例。

- ❑ `numpy_rng`：随机数生成引擎。
- ❑ `theano_rng`：Theano 随机数生成引擎。
- ❑ `input`：输入信号。
- ❑ `n_visible`：输入层尺寸为 28×28 。
- ❑ `n_hidden`：隐藏层神经元数为 500。

第 51~54 行：调用 `DenosingAutoencoder` 类的 `get_cost_update` 方法，得到代价函数定义和参数的更新规则。

第 55~62 行：定义 Theano 函数 `da_net` 完成网络的训练功能。

第 63 行：记录训练开始时间。

第 64 行：循环使用训练样本集进行训练，直到最大训练次数为止。

第 66 行：对于每次循环，对训练样本集上所有迷你批次，执行下面操作。

第 67 行：调用 Theano 函数 `da_net` 计算代价函数值，利用梯度下降算法更新网络参数。

第 68 行：打印本次训练样本集全部训练后的平均误差函数值。

第 69 行：取得网络训练的结束时间。

第 71~73 行：打印汇总信息。

第 74~77 行：将连接权值矩阵绘制成图像方式，便于可视化。在本例中，连接权值矩阵为 784×500 的矩阵，因为图片大小为 $28 \times 28 = 784$ ，而隐藏层的神经元数为 500，所以将连接权值矩阵转置，变为 500×784 的矩阵，每一行有 784 个连接权值，作为 `tile_raster_images` 的第一个参数，`img_shape` 为将第一个参数的一行变为其所规定形状的二维矩阵形式，`tile_shape` 表示会显示几行几列的图像，`tile_spacing` 为图像间的间隔。`tile_raster_images` 函数会返回一个数组，调用 `Image.fromArray` 方法将数组变成一个图像。

第 78 行：将该图像保存为图像文件。

下面我们来看将权值矩阵转换为图像数组的工具函数，由于这部分代码与去噪自动编码器没有关系，就不在此进行详细解析了，但为了内容的完整性，将代码列出来。

```

1 import numpy
2
3 def scale_to_unit_interval(ndar, eps=1e-8):
4     ndar = ndar.copy()
5     ndar -= ndar.min()
6     ndar *= 1.0 / (ndar.max() + eps)
7     return ndar
8
9
10 def tile_raster_images(X, img_shape, tile_shape, tile_spacing=(0, 0),
11                        scale_rows_to_unit_interval=True,
12                        output_pixel_vals=True):
13     assert len(img_shape) == 2
14     assert len(tile_shape) == 2
15     assert len(tile_spacing) == 2
16
17     # The expression below can be re-written in a more C style as
18     # follows :
19     #
20     # out_shape = [0,0]
21     # out_shape[0] = (img_shape[0]+tile_spacing[0])*tile_shape[0] -
22     #                 tile_spacing[0]
23     # out_shape[1] = (img_shape[1]+tile_spacing[1])*tile_shape[1] -
24     #                 tile_spacing[1]
25     out_shape = [
26         (ishp + tsp) * tshp - tsp
27         for ishp, tshp, tsp in zip(img_shape, tile_shape, tile_spacing)
28     ]
29
30     if isinstance(X, tuple):
31         assert len(X) == 4
32         # Create an output numpy ndarray to store the image
33         if output_pixel_vals:
34             out_array = numpy.zeros((out_shape[0], out_shape[1], 4),
35                                    dtype='uint8')
36         else:
37             out_array = numpy.zeros((out_shape[0], out_shape[1], 4),
38                                    dtype=X.dtype)
39
40         #colors default to 0, alpha defaults to 1 (opaque)
41         if output_pixel_vals:
42             channel_defaults = [0, 0, 0, 255]
43         else:
44             channel_defaults = [0., 0., 0., 1.]
45
46         for i in range(4):
47             if X[i] is None:
48                 # if channel is None, fill it with zeros of the correct
49                 # dtype
50                 dt = out_array.dtype
51                 if output_pixel_vals:
52                     dt = 'uint8'
53                 out_array[:, :, i] = numpy.zeros(
54                     out_shape,
55                     dtype=dt
56                 ) + channel_defaults[i]
57             else:
58                 # use a recurrent call to compute the channel and store it
59                 # in the output
60                 out_array[:, :, i] = tile_raster_images(
61                     X[i], img_shape, tile_shape, tile_spacing,
62                     scale_rows_to_unit_interval, output_pixel_vals)
63         return out_array
64
65     else:
66         # if we are dealing with only one channel
67         H, W = img_shape
68         Hs, Ws = tile_spacing
69
70         # generate a matrix to store the output
71         dt = X.dtype
72         if output_pixel_vals:
73             dt = 'uint8'
74         out_array = numpy.zeros(out_shape, dtype=dt)
75
76         for tile_row in range(tile_shape[0]):
77             for tile_col in range(tile_shape[1]):

```

```

78         if tile_row * tile_shape[1] + tile_col < X.shape[0]:
79             this_x = X[tile_row * tile_shape[1] + tile_col]
80             if scale_rows_to_unit_interval:
81                 # if we should scale values to be between 0 and 1
82                 # do this by calling the 'scale_to_unit_interval'
83                 # function
84                 this_img = scale_to_unit_interval(
85                     this_x.reshape(img_shape))
86             else:
87                 this_img = this_x.reshape(img_shape)
88             # add the slice to the corresponding position in the
89             # output array
90             c = 1
91             if output_pixel_vals:
92                 c = 255
93             out_array[
94                 tile_row * (H + Hs): tile_row * (H + Hs) + H,
95                 tile_col * (W + Ws): tile_col * (W + Ws) + W
96             ] = this_img * c
97         return out_array
98
99 def scale_to_unit_interval(ndar, eps=1e-8):
100     ndar = ndar.copy()
101     ndar -= ndar.min()
102     ndar *= 1.0 / (ndar.max() + eps)
103     return ndar
104
105
106 def tile_raster_images(X, img_shape, tile_shape, tile_spacing=(0, 0),
107                        scale_rows_to_unit_interval=True,
108                        output_pixel_vals=True):
109     assert len(img_shape) == 2
110     assert len(tile_shape) == 2
111     assert len(tile_spacing) == 2
112
113     # The expression below can be re-written in a more C style as
114     # follows :
115     #
116     # out_shape = [0,0]
117     # out_shape[0] = (img_shape[0]+tile_spacing[0])*tile_shape[0] -
118     #                 tile_spacing[0]
119     # out_shape[1] = (img_shape[1]+tile_spacing[1])*tile_shape[1] -
120     #                 tile_spacing[1]
121     out_shape = [
122         (ishp + tsp) * tshp - tsp
123         for ishp, tshp, tsp in zip(img_shape, tile_shape, tile_spacing)
124     ]
125
126     if isinstance(X, tuple):
127         assert len(X) == 4
128         # Create an output numpy ndarray to store the image
129         if output_pixel_vals:
130             out_array = numpy.zeros((out_shape[0], out_shape[1], 4),
131                                     dtype='uint8')
132         else:
133             out_array = numpy.zeros((out_shape[0], out_shape[1], 4),
134                                     dtype=X.dtype)
135
136         #colors default to 0, alpha defaults to 1 (opaque)
137         if output_pixel_vals:
138             channel_defaults = [0, 0, 0, 255]
139         else:
140             channel_defaults = [0., 0., 0., 1.]
141
142         for i in range(4):
143             if X[i] is None:
144                 # if channel is None, fill it with zeros of the correct
145                 # dtype
146                 dt = out_array.dtype
147                 if output_pixel_vals:
148                     dt = 'uint8'
149                 out_array[:, :, i] = numpy.zeros(
150                     out_shape,
151                     dtype=dt,
152                 ) + channel_defaults[i]
153             else:
154                 # use a recurrent call to compute the channel and store it
155                 # in the output
156                 out_array[:, :, i] = tile_raster_images(

```

```

157         X[i], img_shape, tile_shape, tile_spacing,
158         scale_rows_to_unit_interval, output_pixel_vals)
159     return out_array
160
161     else:
162         # if we are dealing with only one channel
163         H, W = img_shape
164         Hs, Ws = tile_spacing
165
166         # generate a matrix to store the output
167         dt = X.dtype
168         if output_pixel_vals:
169             dt = 'uint8'
170         out_array = numpy.zeros(out_shape, dtype=dt)
171
172         for tile_row in range(tile_shape[0]):
173             for tile_col in range(tile_shape[1]):
174                 if tile_row * tile_shape[1] + tile_col < X.shape[0]:
175                     this_x = X[tile_row * tile_shape[1] + tile_col]
176                     if scale_rows_to_unit_interval:
177                         # if we should scale values to be between 0 and 1
178                         # do this by calling the `scale_to_unit_interval`
179                         # function
180                         this_img = scale_to_unit_interval(
181                             this_x.reshape(img_shape))
182                     else:
183                         this_img = this_x.reshape(img_shape)
184                     # add the slice to the corresponding position in the
185                     # output array
186                     c = 1
187                     if output_pixel_vals:
188                         c = 255
189                     out_array[
190                         tile_row * (H + Hs): tile_row * (H + Hs) + H,
191                         tile_col * (W + Ws): tile_col * (W + Ws) + W
192                     ] = this_img * c
193         return out_array

```

下面需要载入 MNIST 手写数字识别数据集。我们用 `MnistLoader` 类来完成这个工作，代码如下：

```

1 from __future__ import print_function
2 __docformat__ = 'restructuredtext en'
3 import six.moves.cPickle as pickle
4 import gzip
5 import os
6 import sys
7 import timeit
8 import numpy
9 import theano
10 import theano.tensor as T
11
12 class MnistLoader(object):
13     def load_data(self, dataset):
14         data_dir, data_file = os.path.split(dataset)
15         if data_dir == "" and not os.path.isfile(dataset):
16             new_path = os.path.join(
17                 os.path.split(__file__)[0],
18                 "..",
19                 "data",
20                 dataset
21             )
22             if os.path.isfile(new_path) or data_file == 'mnist.pkl.gz':
23                 dataset = new_path
24
25         if (not os.path.isfile(dataset)) and data_file == 'mnist.pkl.gz':
26             from six.moves import urllib
27             origin = (
28                 'http://www.iro.umontreal.ca/~lisa/deep/data/'
29                 'mnist/mnist.pkl.gz'
30             )
31             print('Downloading data from %s' % origin)
32             urllib.request.urlretrieve(origin, dataset)
33

```

```

34     print('... loading data')
35     # Load the dataset
36     with gzip.open(dataset, 'rb') as f:
37         try:
38             train_set, valid_set, test_set = pickle.load(f,
39                                                         encoding='latin1')
40         except:
41             train_set, valid_set, test_set = pickle.load(f)
42     def shared_dataset(data_xy, borrow=True):
43         data_x, data_y = data_xy
44         shared_x = theano.shared(numpy.asarray(data_x,
45                                                 dtype=theano.config.floatX),
46                                 borrow=borrow)
47         shared_y = theano.shared(numpy.asarray(data_y,
48                                                 dtype=theano.config.floatX),
49                                 borrow=borrow)
50         return shared_x, T.cast(shared_y, 'int32')
51
52     test_set_x, test_set_y = shared_dataset(test_set)
53     valid_set_x, valid_set_y = shared_dataset(valid_set)
54     train_set_x, train_set_y = shared_dataset(train_set)
55
56     rval = [(train_set_x, train_set_y), (valid_set_x, valid_set_y),
57            (test_set_x, test_set_y)]
58     return rval

```

第 13 行：定义数据载入接口，参数为 MNIST 手写数字识别数据集文件。

第 14~33 行：MNIST 手写数字识别数据集文件如果存在则打开该文件，如果不存在则从指定网址下载该文件。

第 52~54 行：将训练样本集、验证样本集、测试样本集定义为 Theano 的共享变量，以便在 Theano 预编译函数中使用。

第 56~58 行：以指定格式返回 MNIST 手写数字识别数据内容。

第 9 章

堆叠自动编码器

在上一章中，我们讨论了去噪自动编码器，并讨论了 Theano 框架实现的细节。在本章中，我们将讨论去噪自动编码器的主要应用，即组成堆叠去噪自动编码器（SdA）。我们将以 MNIST 手写字母识别为例，讲解怎么用堆叠去噪自动编码器来解决这一问题。

堆叠去噪自动编码器是由一系列去噪自动编码器堆叠而成的，每个去噪自动编码器的中间层（编码层）作为下一层的输入层，这样一层一层堆叠起来构成一个深层网络，这些网络组成堆叠去噪自动编码器的表示部分。这部分通过非监督学习，逐层进行培训，每一层均可以还原加入随机噪声后的输入信号，而此时在每个去噪自动编码器中间层的输出信号，可以视为对原始输入信号的某种简化表示。

当将所有去噪自动编码器堆叠形成的网络训练完成之后，首先把最后一层的中间层接入逻辑回归网络，作为其输入层，这样就形成了一个新的多层 BP 网络，隐藏层之间的权值，就是前面利用去噪自动编码器逐层训练时得到的连接权值矩阵。然后将这个网络视为一个标准的 BP 网络，利用原来的 BP 网络算法进行监督学习，以达到我们想要的状态。

可能读者会有疑问，为什么不直接使用多层 BP 网络呢？其实，在 BP 网络诞生之初，就有人基于它做具有多个隐藏层的深度网络了。但是人们很快就发现，基于误差反向传播的 BP 网络，利用随机梯度下降算法来调整权值，但是随着层数的加深，离输出层较远的隐藏层的权值调整量将递减，最后导致这种深度网络学习速度非常慢，直接限制了它的使用。因此，在深度学习崛起之前，深层网络基本没有成功的应用案例。

从堆叠去噪自动编码器来看，我们首先通过逐层非监督学习方式训练独立的去噪自动编码器，可以视为神经网络自动发现问题域的特征的过程，通过自动特征提取，找到解决问题的最优特征。而堆叠去噪自动编码器的训练，可以视为已经对多层 BP 网络进行了初步训练，最后的监督学习是对网络权值的微调优化，这样可以较好地解决深度前馈网络学习收敛速度慢的问题，使其具有实用价值。

9.1 堆叠去噪自动编码器

堆叠去噪自动编码器是基于去噪自动编码器构建的，所以本章有些代码与上一章相同，但是为了代码的完整性，我们会将相关代码再列出来，使读者无须参考上一章内容也可以很好地理解本章的内容。

我们通过定义去噪自动编码器类来实现这些功能，代码如下：

```
1 from __future__ import print_function
2 import os
3 import sys
4 import timeit
5 import numpy
6 import theano
7 import theano.tensor as T
8 from theano.tensor.shared_randomstreams import RandomStreams
9
10 class DenoisingAutoencoder(object):
11     def __init__(
12         self,
13         numpy_rng,
14         theano_rng=None,
15         input=None,
16         n_visible=784,
17         n_hidden=500,
18         W=None,
19         bhid=None,
20         bvis=None
21     ):
22         self.n_visible = n_visible
23         self.n_hidden = n_hidden
24         if not theano_rng:
25             theano_rng = RandomStreams(numpy_rng.randint(2 ** 30))
26         if not W:
27             initial_W = numpy.asarray(
28                 numpy_rng.uniform(
29                     low=-4 * numpy.sqrt(6. / (n_hidden + n_visible)),
30                     high=4 * numpy.sqrt(6. / (n_hidden + n_visible)),
31                     size=(n_visible, n_hidden)
32                 ),
33                 dtype=theano.config.floatX
34             )
35             W = theano.shared(value=initial_W, name='W', borrow=True)
36
37         if not bvis:
38             bvis = theano.shared(
39                 value=numpy.zeros(
40                     n_visible,
41                     dtype=theano.config.floatX
42                 ),
43                 borrow=True
44             )
45
46         if not bhid:
47             bhid = theano.shared(
48                 value=numpy.zeros(
49                     n_hidden,
50                     dtype=theano.config.floatX
51                 ),
52                 name='b',
53                 borrow=True
54             )
55         self.W = W
56         self.b = bhid
57         self.b_prime = bvis
58         self.W_prime = self.W.T
59         self.theano_rng = theano_rng
60         if input is None:
61             self.x = T.dmatrix(name='input')
```

```

62     else:
63         self.x = input
64         self.params = [self.W, self.b, self.b_prime]
65
66     def get_corrupted_input(self, input, corruption_level):
67         return self.theano_rng.binomial(size=input.shape, n=1,
68                                         p=1 - corruption_level,
69                                         dtype=theano.config.floatX) * input
70
71     def get_hidden_values(self, input):
72         return T.nnet.sigmoid(T.dot(input, self.W) + self.b)
73
74     def get_reconstructed_input(self, hidden):
75         return T.nnet.sigmoid(T.dot(hidden, self.W_prime) + self.b_prime)
76
77     def get_cost_updates(self, corruption_level, learning_rate):
78         tilde_x = self.get_corrupted_input(self.x, corruption_level)
79         y = self.get_hidden_values(tilde_x)
80         z = self.get_reconstructed_input(y)
81         L = - T.sum(self.x * T.log(z) + (1 - self.x) * T.log(1 - z), axis=1)
82         cost = T.mean(L)
83         gparams = T.grad(cost, self.params)
84         updates = [
85             (param, param - learning_rate * gparam)
86             for param, gparam in zip(self.params, gparams)
87         ]
88
89     return (cost, updates)

```

第 10 行：定义 DenosingAutoencoder 类。

第 11 行：定义 DenosingAutoencoder 类构造函数。

- ☐ numpy_rng: 随机数生成引擎，用于初始化权值。
- ☐ theano_rng: Theano 的随机数生成器。
- ☐ input: 输入信号，使用去噪自动编码器时其为空，使用堆叠自动编码器时其为 Theano 变量，由下层计算结果作为输入。
- ☐ n_visible: 输入层神经元个数。
- ☐ W: 连接权值。
- ☐ bhid: 偏移值 Bias，单独去噪自动编码器时可忽略。
- ☐ bvis: 偏移值 Bias，单独去噪自动编码器时可忽略。

第 22 行：设置可见层神经元数量。

第 23 行：设置隐藏层神经元数量。

第 24、25 行：如果没有指定 Theano 随机数生成引擎，利用 theano_rng 重新生成一个。

第 26~35 行：如果没有指定连接权值矩阵，则用随机数进行初始化，并将其定义为 Theano 的共享变量 W。

第 37~44 行：如果没有指定可见层偏移量 bvis，则用 0 初始化。

第 46~54 行：如果没有指定隐藏层偏移量 bhid，则用 0 初始化。

第 55 行：设置类属性 W 为之前定义的共享变量连接权值 W。

第 56 行：设置类属性 b 为隐藏层偏移量 bhid。

第 57 行：设置类属性 b_prime 为可见层偏移量 bvis。

第 58 行：设置类属性 w_prime 为隐藏层到输出层连接权值，其值为输入层到隐藏层连接权值的转置。

第 59 行：随机数生成引擎，设置类属性 theano_rng。

第 60~63 行：设置类属性 x 为输入信号，如果未指定则初始化为一个矩阵。

第 64 行：设置模型参数为连接权值、隐藏层偏移量和可见层偏移量。

第 66~69 行：定义获取加入噪声后输入信号的方法，调用 Theano 方法向输入信号中加入随机噪声。

第 71、72 行：获取隐藏层输入值。

第 74、75 行：获取输出层输出值。

第 77 行：定义计算代价函数和更新参数的方法。

第 78 行：获取加入随机噪声的输入信号。

第 79 行：获取隐藏层的输出信号。

第 80 行：获取输出层的重建信号。

第 81 行：计算总的代价函数值，公式为： $x \cdot \log \hat{x} + (1 - x) \log(1 - \hat{x})$ 。

第 82 行：计算代价函数的平均值。

第 83 行：将代价函数对所有参数求导。

第 84~87 行：定义参数更新规则为 $w = w - \alpha \frac{\partial \mathcal{L}}{\partial w}$ 。

下面我们来定义堆叠去噪自动编码器类。

根据上一节的讨论，我们在逐层训练完去噪自动编码器之后，先将其进行堆叠，形成一个标准的多层感知器模型，再用标准 BP 算法进行微调。所以需要隐藏层和逻辑回归层来完成这些功能。

首先来定义隐藏层 HiddenLayer 类，代码如下：

```
1 from __future__ import print_function
2 __docformat__ = 'restructuredtext en'
3 import os
4 import sys
5 import timeit
6 import numpy
7 import theano
8 import theano.tensor as T
9 from logistic_regression import LogisticRegression
10
11 # start-snippet-1
12 class HiddenLayer(object):
13     def __init__(self, rng, input, n_in, n_out, W=None, b=None,
14                 activation=T.tanh):
15         self.input = input
16         if W is None:
17             W_values = numpy.asarray(
18                 rng.uniform(
19                     low=-numpy.sqrt(6. / (n_in + n_out)),
20                     high=numpy.sqrt(6. / (n_in + n_out)),
21                     size=(n_in, n_out)
22                 ),
23                 dtype=theano.config.floatX
24             )
25             if activation == theano.tensor.nnet.sigmoid:
26                 W_values *= 4
27
28             W = theano.shared(value=W_values, name='W', borrow=True)
29
30         if b is None:
31             b_values = numpy.zeros((n_out,), dtype=theano.config.floatX)
32             b = theano.shared(value=b_values, name='b', borrow=True)
33
```

```

34     self.W = W
35     self.b = b
36
37     lin_output = T.dot(input, self.W) + self.b
38     self.output = (
39         lin_output if activation is None
40         else activation(lin_output)
41     )
42     # parameters of the model
43     self.params = [self.W, self.b]

```

第 12 行：定义 `HiddenLayer` 类。

第 13、14 行：定义 `HiddenLayer` 类的构造函数。

- ❑ `rng`: 随机数生成引擎。
- ❑ `input`: 输入信号。
- ❑ `n_in`: 输入信号维度。
- ❑ `n_out`: 下一层神经元数量。
- ❑ `W`: 上一层到本层的连接权值矩阵，默认时空。
- ❑ `b`: 本层神经元的偏移量，默认时空。
- ❑ `activation`: 激活函数，默认时为双曲正切函数。

第 15 行：设置输入信号为参数中的输入信号。

第 16~28 行：如果没有提供参数 `W`，则以随机数初始化连接权值矩阵，将连接权值矩阵定义为 `Theano` 共享变量。

第 30~32 行：如果没有提供参数 `b`，则以 0 初始化偏移值，并将其定义为 `Theano` 共享变量。

第 34 行：将连接权值赋给类属性 `W`。

第 35 行：将偏移值赋给类属性 `b`。

第 37 行：定义本层神经元的输入信号线性。

第 38~41 行：如果没有定义激活函数，则直接输出线性，否则应用激活函数求输出值。

第 43 行：定义本层参数为连接权值和偏移量 `Bias`。

由于我们的微调网络是一个标准的多层感知器模型，所以需要定义一个多层感知器类 `MLP`，代码如下：

```

1 from __future__ import print_function
2 __docformat__ = 'restructuredtext en'
3 import os
4 import sys
5 import timeit
6 import numpy
7 import theano
8 import theano.tensor as T
9 from logistic_regression import LogisticRegression
10 from mnist_loader import MnistLoader
11 from hidden_layer import HiddenLayer
12
13 class MLP(object):
14     def __init__(self, rng, input, n_in, n_hidden, n_out):
15         self.hiddenLayer = HiddenLayer(
16             rng=rng,
17             input=input,
18             n_in=n_in,
19             n_out=n_hidden,
20             activation=T.tanh

```

```

21     )
22     self.logRegressionLayer = LogisticRegression(
23         input=self.hiddenLayer.output,
24         n_in=n_hidden,
25         n_out=n_out
26     )
27     self.L1 = (
28         abs(self.hiddenLayer.W).sum()
29         + abs(self.logRegressionLayer.W).sum()
30     )
31     self.L2_sqr = (
32         (self.hiddenLayer.W ** 2).sum()
33         + (self.logRegressionLayer.W ** 2).sum()
34     )
35     self.negative_log_likelihood = (
36         self.logRegressionLayer.negative_log_likelihood
37     )
38     self.errors = self.logRegressionLayer.errors
39     self.params = self.hiddenLayer.params + \
40                 self.logRegressionLayer.params
41     self.input = input

```

第 13 行：定义 MLP 类。

第 14 行：定义 MLP 类构造函数。

- ❑ `rng`: 随机数生成引擎。
- ❑ `input`: 输入向量。
- ❑ `n_in`: 输入向量维度，也是输入层神经元数目。
- ❑ `n_hidden`: 隐藏层神经元数目。
- ❑ `n_out`: 输出层神经元数目，也是模式分类问题中的类别数目。

第 15~21 行：定义隐藏层，其输入为网络的输入，输入向量维度为 `n_in`，隐藏层神经元数目为 `n_hidden`，激活函数为双曲正切函数。

第 22~26 行：定义输出层，其输入为隐藏层的输出，输入向量维度为隐藏层神经元数目，本层神经元数目为 `n_out`，即模式分类的类别数。

第 27~30 行：定义调整量 `L1`，其定义为所有连接权值绝对值之和。

第 31~34 行：定义调整量 `L2`，其定义为所有连接权值平方之和。

`L1` 和 `L2` 是代价函数的附加项，保证在优化过程中优先取连接权值小的解决方案。

第 35~37 行：定义代价函数为逻辑回归模型的负对数似然函数。

第 38 行：定义逻辑回归层的误差为多层感知器模型的误差。

第 39、40 行：定义参数为连接权值加本层偏移量 `Bias`。

第 41 行：输入信号为参数中的输入信号。

最后的分类层采用 `softmax` 激活函数的逻辑回归层，定义逻辑回归层 `LogisticRegression` 类的代码如下：

```

1 from __future__ import print_function
2 __docformat__ = 'restructuredtext en'
3 import six.moves.cPickle as pickle
4 import gzip
5 import os
6 import sys
7 import timeit
8 import numpy
9 import theano
10 import theano.tensor as T
11
12 class LogisticRegression(object):
13     def __init__(self, input, n_in, n_out):

```

```

14     self.W = theano.shared(
15         value=numpy.zeros(
16             (n_in, n_out),
17             dtype=theano.config.floatX
18         ),
19         name='W',
20         borrow=True
21     )
22     self.b = theano.shared(
23         value=numpy.zeros(
24             (n_out,),
25             dtype=theano.config.floatX
26         ),
27         name='b',
28         borrow=True
29     )
30     self.p_y_given_x = T.nnet.softmax(T.dot(input, self.W) + self.b)
31     self.y_pred = T.argmax(self.p_y_given_x, axis=1)
32     self.params = [self.W, self.b]
33     self.input = input
34     print("Yantao: *****")
35
36     def negative_log_likelihood(self, y):
37         return -T.mean(T.log(self.p_y_given_x)[T.arange(y.shape[0]), y])
38
39     def errors(self, y):
40         if y.ndim != self.y_pred.ndim:
41             raise TypeError(
42                 'y should have the same shape as self.y_pred',
43                 ('y', y.type, 'y_pred', self.y_pred.type)
44             )
45         if y.dtype.startswith('int'):
46             return T.mean(T.neq(self.y_pred, y))
47         else:
48             raise NotImplementedError()

```

第 12 行：定义 LogisticRegression 类。

第 13 行：定义 LogisticRegression 类构造函数。

- ❑ input: 输入向量。
- ❑ n_in: 输入信号维度。
- ❑ n_out: 分类类别数。

第 14~21 行：如果没有定义连接权值矩阵，则用 0 进行初始化，并定义为 Theano 共享变量。

第 22~29 行：如果没有定义偏移量 Bias，则用 0 进行初始化，并定义为 Theano 共享变量。

第 30 行：激活函数为线性和的 softmax 函数。

第 31 行：定义输出值为 softmax 函数值最大的一个类别。

第 32 行：定义本层参数为连接权值矩阵和偏移量 Bias。

第 33 行：定义类属性 input 为参数的 input。

第 36、37 行：定义本层（也是整个网络）的代价函数为负对数似然函数。

第 39~48 行：定义错误处理函数。

下面来定义堆叠去噪自动编码器 SdA 类，这是整个网络的核心类，代码如下：

```

1 from __future__ import print_function
2 import os
3 import sys
4 import timeit
5 import numpy
6 import theano
7 import theano.tensor as T
8 from theano.tensor.shared_randomstreams import RandomStreams

```

```

9 from logistic_regression import LogisticRegression
10 from hidden_layer import HiddenLayer
11 from denosing_autoencoder import DenosingAutoencoder
12
13 class SdA(object):
14     def __init__(
15         self,
16         numpy_rng,
17         theano_rng=None,
18         n_ins=784,
19         hidden_layers_sizes=[500, 500],
20         n_outs=10,
21         corruption_levels=[0.1, 0.1]
22     ):
23         self.sigmoid_layers = []
24         self.dA_layers = []
25         self.params = []
26         self.n_layers = len(hidden_layers_sizes)
27
28         assert self.n_layers > 0
29
30         if not theano_rng:
31             theano_rng = RandomStreams(numpy_rng.randint(2 ** 30))
32         self.x = T.matrix('x') # the data is presented as rasterized images
33         self.y = T.ivector('y') # the labels are presented as 1D vector of
34         for i in range(self.n_layers):
35             if i == 0:
36                 input_size = n_ins
37             else:
38                 input_size = hidden_layers_sizes[i - 1]
39             if i == 0:
40                 layer_input = self.x
41             else:
42                 layer_input = self.sigmoid_layers[-1].output
43
44             sigmoid_layer = HiddenLayer(rng=numpy_rng,
45                                         input=layer_input,
46                                         n_in=input_size,
47                                         n_out=hidden_layers_sizes[i],
48                                         activation=T.nnet.sigmoid)
49             self.sigmoid_layers.append(sigmoid_layer)
50             self.params.extend(sigmoid_layer.params)
51             dA_layer = DenosingAutoencoder(numpy_rng=numpy_rng,
52                                           theano_rng=theano_rng,
53                                           input=layer_input,
54                                           n_visible=input_size,
55                                           n_hidden=hidden_layers_sizes[i],
56                                           W=sigmoid_layer.W,
57                                           bhid=sigmoid_layer.b)
58             self.dA_layers.append(dA_layer)
59             self.logLayer = LogisticRegression(
60                 input=self.sigmoid_layers[-1].output,
61                 n_in=hidden_layers_sizes[-1],
62                 n_out=n_outs
63             )
64             self.params.extend(self.logLayer.params)
65             self.finetune_cost = self.logLayer.negative_log_likelihood(self.y)
66             self.errors = self.logLayer.errors(self.y)
67
68     def pretraining_functions(self, train_set_x, batch_size):
69         index = T.scalar('index') # index to a minibatch
70         corruption_level = T.scalar('corruption') # % of corruption to use
71         learning_rate = T.scalar('lr') # learning rate to use
72         batch_begin = index * batch_size
73         batch_end = batch_begin + batch_size
74         pretrain_fns = []
75         for dA in self.dA_layers:
76             cost, updates = dA.get_cost_updates(corruption_level,
77                                                  learning_rate)
78             fn = theano.function(
79                 inputs=[
80                     index,
81                     theano.In(corruption_level, value=0.2),
82                     theano.In(learning_rate, value=0.1)
83                 ],
84                 outputs=cost,

```

```

85         updates=updates,
86         givens={
87             self.x: train_set_x[batch_begin: batch_end]
88         }
89     )
90     pretrain_fns.append(fn)
91     return pretrain_fns
92
93     def build_finetune_functions(self, datasets, batch_size, learning_rate):
94         (train_set_x, train_set_y) = datasets[0]
95         (valid_set_x, valid_set_y) = datasets[1]
96         (test_set_x, test_set_y) = datasets[2]
97         n_valid_batches = valid_set_x.get_value(borrow=True).shape[0]
98         n_valid_batches //= batch_size
99         n_test_batches = test_set_x.get_value(borrow=True).shape[0]
100        n_test_batches //= batch_size
101        index = T.lscalar('index')
102        gparams = T.grad(self.finetune_cost, self.params)
103        updates = [
104            (param, param - gparam * learning_rate)
105            for param, gparam in zip(self.params, gparams)
106        ]
107        train_fn = theano.function(
108            inputs=[index],
109            outputs=self.finetune_cost,
110            updates=updates,
111            givens={
112                self.x: train_set_x[
113                    index * batch_size: (index + 1) * batch_size
114                ],
115                self.y: train_set_y[
116                    index * batch_size: (index + 1) * batch_size
117                ]
118            },
119            name='train'
120        )
121        test_score_i = theano.function(
122            [index],
123            self.errors,
124            givens={
125                self.x: test_set_x[
126                    index * batch_size: (index + 1) * batch_size
127                ],
128                self.y: test_set_y[
129                    index * batch_size: (index + 1) * batch_size
130                ]
131            },
132            name='test'
133        )
134        valid_score_i = theano.function(
135            [index],
136            self.errors,
137            givens={
138                self.x: valid_set_x[
139                    index * batch_size: (index + 1) * batch_size
140                ],
141                self.y: valid_set_y[
142                    index * batch_size: (index + 1) * batch_size
143                ]
144            },
145            name='valid'
146        )
147        def valid_score():
148            return [valid_score_i(i) for i in range(n_valid_batches)]
149        def test_score():
150            return [test_score_i(i) for i in range(n_test_batches)]
151        return train_fn, valid_score, test_score

```

第13行：定义 SdA 类。

第14~22行：定义 SdA 类构造函数。

- ☐ numpy_rng: numpy 的随机数生成引擎。
- ☐ theano_rng: theano 的随机数生成引擎。

- ❑ `n_ins`: 输入层维度, 默认值为 784 (28×28)。
- ❑ `hidden_layers_sizes`: 隐藏层大小, 共两层, 神经元数均为 500。
- ❑ `n_outs`: 输出层神经元数量, 也是类别数量。
- ❑ `corruption_levels`: 去噪自动编码器的噪声比例, 在本实例中, 共有两个堆叠的去噪自动编码器, 第一个去噪自动编码器的中间层与第二个去噪自动编码器的输入层共用, 这两层与隐藏层也是相同的。

第 23 行: 定义类属性 `sigmoid_layers` 列表, 因为在本例中, 当作为多层感知器模型进行微调时, 隐藏层采用 Sigmoid 激活函数。

第 24 行: 定义类属性 `dA_layers` 列表, 管理组成的去噪自动编码器。

第 25 行: 定义类属性 `params` 参数列表。

第 26 行: 定义类属性 `n_layers`, 表示共有几个隐藏层。

第 30、31 行: 初始化 `theano_rng`, `theano` 的随机数生成引擎。

第 32 行: 将样本输入信号部分组成输入信号矩阵, 即设计矩阵。

第 33 行: 将样本输出标签作为输出向量。

第 34 行: 对于网络的每一层执行循环 (第 35~58 行)。

第 35、36 行: 当为最底下一层时, 本层的输入信号即网络的输入信号维度。

第 38 行: 如果不是最底下一层, 则输入信号维度为上一个隐藏层的神经元数量。

第 39、40 行: 当为最底下一层时, 本层的输入信号即网络的输入信号。

第 42 行: 如果不是最底下一层, 则输入信号为上一个隐藏层的输出信号。

第 44~48 行: 初始化隐藏层。

- ❑ `rng`: 由参数 `numpy` 指定。
- ❑ `input`: 输入信号为本层的输入信号。
- ❑ `n_in`: 输入信号维度。
- ❑ `n_out`: 神经元数量。
- ❑ `activation`: 激活函数为 Sigmoid 函数。

第 49 行: 将新生成的 `sigmoid_layer` 添加到隐藏层列表中。

第 50 行: 将新建隐藏层参数加到本网络参数集中。

第 51~57 行: 初始化去噪自动编码器层。

- ❑ `numpy_rng`: 由参数 `numpy_rng` 指定。
- ❑ `theano_rng`: 由参数 `theano_rng` 指定。
- ❑ `input`: 本层的输入信号。
- ❑ `n_hidden`: 指定神经元数量。
- ❑ `W`: 连接权值矩阵与前面生成的隐藏层共享。
- ❑ `b`: 偏移量 Bias 与前面生成的隐藏层共享。

第 58 行: 将新生成的去噪自动编码器加入去噪自动编码器列表中。

第 59~63 行: 初始化逻辑回归层。

- ❑ `input`: 最后一个隐藏层的输出。

❑ `n_in`: 最后一个隐藏层的神经元个数。

❑ `n_out`: 分类类别数。

第 64 行: 将逻辑回归层参数加入到整个网络的参数集中。

第 65 行: 作为多层感知器模型微调时, 代价函数为逻辑回归层的代价函数。

第 66 行: 定义网络整体误差为逻辑回归层误差。

第 68 行: 定义预训练方法。

❑ `train_set_x`: 训练样本集输入部分。

❑ `batch_size`: 迷你批次大小。

第 69 行: 定义 Theano 变量 `index` 代表迷你批次索引号。

第 70 行: 定义 Theano 变量表示去噪自动编码器添加噪声水平。

第 71 行: 定义 Theano 变量表示学习率。

第 72 行: 计算迷你批次开始位置。

第 73 行: 计算迷你批次结束位置。

第 74 行: 定义预培训去噪自动编码器 Theano 函数列表。

第 75 行: 对每个去噪自动编码器层循环。

第 76、77 行: 以噪声水平和学习率为参数, 获取 `DenosingAutoencoder` 类中的代价函数和参数更新规则。

第 78~89 行: 定义 Theano 函数来表示单个去噪自动编码器。

❑ `input`: 输入信号, 包括噪声水平和学习率。

❑ `output`: 输出值为代价函数。

❑ `updates`: 参数更新规则。

❑ 通过 Theano 的 `given` 关键字, 指定运行时本函数使用的实际参数。

第 90 行: 将新定义的 Theano 函数添加到预培训的 Theano 函数列表中。

第 91 行: 处理完所有去噪自动编码器层后, 将预培训的 Theano 函数列表返回。

第 93 行: 定义创建进行微调的多层感知器网络方法。

❑ `datasets`: 由 `MnistLoader` 类取出的数据集数据。

❑ `batch_size`: 迷你批次大小。

❑ `learning_rate`: 学习率。

第 94 行: 取出训练样本集输入数据和结果标签。

第 95 行: 取出验证样本集输入数据和结果标签。

第 96 行: 取出测试样本集输入数据和结果标签。

第 97、98 行: 求出验证样本集上迷你批次数量。

第 99、100 行: 求出测试样本集上迷你批次数量。

第 101 行: `index` 为迷你批次上的序号。

第 102 行: 对微调网络的代价函数和网络参数求导。

第 103~106 行: 定义参数更新规则 $w = w - \alpha \frac{\partial \mathcal{L}}{\partial w}$ 。

第 107~120 行：定义 Theano 函数 `train_fn`。

- ❑ `inputs`：输入以 `index` 为序号的迷你批次。
- ❑ `outputs`：微调网络的代价函数。
- ❑ 通过 Theano 的 `givens` 关键字，指定实际运行时输入为训练样本集当前迷你批次输入，输出为训练样本集当前迷你批次结果标签输出。

第 121~133 行：定义 Theano 函数 `test_score_i`，计算测试样本集指定迷你批次上的误差率。

- ❑ `input`：测试样本集迷你批次。
- ❑ `output`：误差函数。
- ❑ 通过 Theano 的 `givens` 关键字，指定实际运行时参数为测试样本集当前迷你批次输入，输出为测试样本集当前迷你批次结果标签输出。

第 134~146 行：定义 Theano 函数 `valid_score_i`，计算验证样本集指定迷你批次上的误差率。

- ❑ `input`：验证样本集迷你批次。
- ❑ `output`：误差函数。
- ❑ 通过 Theano 的 `givens` 关键字，指定实际运行时参数为验证样本集当前迷你批次输入，输出为验证样本集当前迷你批次结果标签输出。

第 147、148 行：定义函数计算验证样本集上所有迷你批次上的误差，并以列表形式返回。

第 149、150 行：定义函数计算测试样本集上所有迷你批次上的误差，并以列表形式返回。

下面介绍堆叠去噪自动编码器引擎 `SdAEngine` 类，这个类中比较特别的部分是其训练方法。训练方法由两部分组成，第一部分是对单个去噪自动编码器的预训练，第二部分是对多层感知器的微调网络进行微调训练，代码如下：

```
1 from __future__ import print_function
2 import os
3 import sys
4 import timeit
5 import numpy
6 import theano
7 import theano.tensor as T
8 from theano.tensor.shared_randomstreams import RandomStreams
9 from mnist_loader import MnistLoader
10 from mlp import HiddenLayer
11 from sda import SdA
12
13 class SdAEngine(object):
14     def __init__(self):
15         print('create SdAEngine')
16
17     def train(finetime_lr=0.1, pretraining_epochs=15,
18             pretrain_lr=0.001, training_epochs=1000,
19             dataset='mnist.pkl.gz', batch_size=1):
20         loader = MnistLoader()
21         datasets = loader.load_data(dataset)
22         train_set_x, train_set_y = datasets[0]
23         valid_set_x, valid_set_y = datasets[1]
24         test_set_x, test_set_y = datasets[2]
25         n_train_batches = train_set_x.get_value(borrow=True).shape[0]
26         n_train_batches //= batch_size
27         numpy_rng = numpy.random.RandomState(89677)
28         print('... building the model')
29         sda = SdA(
30             numpy_rng=numpy_rng,
31             n_ins=28 * 28,
```

```

32     hidden_layers_sizes=[1000, 1000, 1000],
33     n_outs=10
34 )
35 print('... getting the pretraining functions')
36 pretraining_fns = sda.pretraining_functions(train_set_x=train_set_x,
37                                             batch_size=batch_size)
38 print('... pre-training the model')
39 start_time = timeit.default_timer()
40 corruption_levels = [.1, .2, .3]
41 for i in range(sda.n_layers):
42     for epoch in range(pretraining_epochs):
43         c = []
44         for batch_index in range(n_train_batches):
45             c.append(pretraining_fns[i](index=batch_index,
46                                       corruption=corruption_levels[i],
47                                       lr=pretrain_lr))
48         print('Pre-training layer %i, epoch %d, cost %f' % (i, \
49                                                         epoch, numpy.mean(c)))
50 end_time = timeit.default_timer()
51 print(('The pretraining code for file ' +
52       os.path.split(__file__)[1] +
53       ' ran for %.2fm' % ((end_time - start_time) / 60.)), \
54       file=sys.stderr)
55 print('... getting the finetuning functions')
56 train_fn, validate_model, test_model = sda.build_finetune_functions(
57     datasets=datasets,
58     batch_size=batch_size,
59     learning_rate=finetune_lr
60 )
61 print('... finetunning the model')
62 patience = 10 * n_train_batches
63 patience_increase = 2.
64 improvement_threshold = 0.995
65 validation_frequency = min(n_train_batches, patience // 2)
66 best_validation_loss = numpy.inf
67 test_score = 0.
68 start_time = timeit.default_timer()
69 done_loopping = False
70 epoch = 0
71 while (epoch < training_epochs) and (not done_loopping):
72     epoch = epoch + 1
73     for minibatch_index in range(n_train_batches):
74         minibatch_avg_cost = train_fn(minibatch_index)
75         iter = (epoch - 1) * n_train_batches + minibatch_index
76         if (iter + 1) % validation_frequency == 0:
77             validation_losses = validate_model()
78             this_validation_loss = numpy.mean(validation_losses)
79             print('epoch %i, minibatch %i/%i,
80                 validation error %f %f' %
81                 (epoch, minibatch_index + 1, n_train_batches,
82                  this_validation_loss * 100.))
83             if this_validation_loss < best_validation_loss:
84                 if (
85                     this_validation_loss < best_validation_loss *
86                     improvement_threshold
87                 ):
88                     patience = max(patience, \
89                                   iter * patience_increase)
89                     best_validation_loss = this_validation_loss
90                     best_iter = iter
91                     test_losses = test_model()
92                     test_score = numpy.mean(test_losses)
93                     print(('epoch %i, minibatch %i/%i, \
94                         test error of '
95                         'best model %f %f') %
96                         (epoch, minibatch_index + 1, n_train_batches,
97                          test_score * 100.))
99                 if patience <= iter:
100                     done_loopping = True
101                     break
102 end_time = timeit.default_timer()
103 print(
104     (
105         'Optimization complete with best validation \
106         score of %f %f, '
107         'on iteration %i, '

```

```

108         'with test performance %f %%'
109     )
110     % (best_validation_loss * 100., best_iter + 1, \
111        test_score * 100.)
112 )
113 print(('The training code for file ' +
114       os.path.split(__file__)[1] +
115       ' ran for %.2fm' % ((end_time - start_time) / 60.)), \
116       file=sys.stderr)

```

第 13、14 行：定义 SdAEngine 类及构造函数。

第 17~19 行：定义网络训练方法。

- ☐ finetune_lr: 网络微调时的学习率。
- ☐ pretraining_epochs: 预训练时训练样本集最大遍历次数。
- ☐ pretrain_lr: 预训练学习率。
- ☐ training_epochs: 微调训练阶段训练样本集最大遍历次数。
- ☐ dataset: 经过预处理的 MNIST 数据集文件名。
- ☐ batch_size: 迷你批次大小。

第 20、21 行：初始化 MNIST 数据集装入类，并装入 MNIST 数据集数据。

第 22 行：取出训练样本集输入数据和输出结果标签数据。

第 23 行：取出验证样本集输入数据和输出结果标签数据。

第 24 行：取出测试样本集输入数据和输出结果标签数据。

第 25、26 行：计算训练样本集迷你批次数量。

第 27 行：生成 numpy 随机数生成引擎。

第 28~34 行：生成堆叠去噪自动编码器类实例。

- ☐ numpy_rng: 刚生成的 numpy 随机数生成引擎。
- ☐ n_ins: 输入信号维度为 784 (28×28)。
- ☐ hidden_layers_sizes: 共有三个隐藏层，每层有 1000 个神经元。
- ☐ n_outs: 分类类别数。

第 36、37 行：网络预训练函数为在堆叠去噪自动编码器中定义的预训练函数，参数为训练样本集输入信号和迷你批次大小。

第 39 行：记录程序开始时间。

第 40 行：由于网络在微调阶段有三个隐藏层，因此网络同样由三个去噪自动编码器构成，这里规定三个去噪自动编码器输入层的噪声水平。

第 41 行：逐层预训练每个去噪自动编码器层。

第 42 行：对每个去噪自动编码器层，迭代次数为预训练时训练样本集最大遍历次数。

第 43 行：定义数组用于保存训练样本集每个迷你批次所对应的代价函数值。

第 44 行：对训练样本集上每个迷你批次循环。

第 45~47 行：调用在 SdA 类中定义的本层所对应的预训练函数。

- ☐ 当前是第几层。
- ☐ corruption: 输入层噪声水平。
- ☐ lr: 学习率。

预训练函数返回值为代价函数的值，将其加入第 43 行定义的变量 c 中。

第 48、49 行：每完成一次训练样本集迭代，打印训练层数、训练样本集迭代次数、平均代价函数值。

第 50 行：完成所有去噪自动编码器层预训练后，记录结束时间。

第 51~54 行：打印网络预训练汇总信息。

第 56~60 行：获取 SdA 类中定义的训练函数、验证模型、测试模型。

❑ `datasets`：由 `MnistLoader` 装入的数据集数据。

❑ `batch_size`：迷你批次大小。

❑ `learning_rate`：微调阶段学习率。

第 62 行：`patience` 允许验证样本集上平均代价函数值，最大多少次迷你批次循环后，仍然没有显著改进的最大循环次数。

第 63 行：在验证样本集上取得最佳平均代价函数值后，变量 `patience` 的调整幅度。

第 64 行：验证样本集上取得最佳平均代价函数值显著改善的标准。

第 65 行：在验证样本集上求平均代价函数的频率。

第 66 行：保存验证样本集上最佳平均代价函数值的变量。

第 67 行：`test_score` 保存在测试样本集上的误差率。

第 68 行：记录微调程序开始时间。

第 69 行：是否需要停止训练程序，初始值为假。

第 70 行：`epoch` 为遍历训练样本集的次数。

第 71 行：当 `epoch` 小于训练样本集最大遍历次数，且是否停止训练程序变量为假时循环执行循环体内容。

第 72 行：训练样本集遍历次数加 1。

第 73 行：循环训练样本集每个迷你批次。

第 74 行：运行微调网络训练函数，并返回平均代价函数值。

第 75 行：求出运行迷你批次总数量。

第 76 行：判断是否需要检查验证样本集上平均代价函数值是否有改进。

第 77 行：如果需要检查，则通过验证模型求出验证样本集上代价函数值之和。

第 78 行：求出验证样本集上代价函数值的平均值。

第 79~82 行：打印验证样本集上代价函数值的平均值的基本信息。

第 83 行：新求出来的验证样本集上代价函数值的平均值是否小于之前记录的验证样本集上代价函数值的平均值的最佳值。

第 84~87 行：如果小于，则检查是否有明显改善（提高 0.5%）。

第 88、89 行：更新允许多少次迷你批次循环验证样本集上代价函数值的平均值没有改进的标准。

第 90 行：更新在验证样本集上代价函数值的平均值的最佳值。

第 91 行：记录取得最佳值时迷你批次遍历次数。

第 92 行：通过测试模型计算测试样本集中代价函数值之和。

第 93 行：求出测试样本集中代价函数值的平均值。

第 94~98 行：打印本次循环的汇总信息。

第 99 行：判断是否足够长时间验证样本集上代价函数值的平均值没有改进。

第 100、101 行：设置终止训练过程变量为真，终止训练过程。

第 102 行：求出训练结束时间。

第 103~116 行：打印本次训练过程汇总信息。

9.2 TensorFlow 实现

完成了上节的准备工作之后，就可以开始将堆叠去噪自动编码器用于 MNIST 手写数字识别数据集上的手写数字识别任务了。

堆叠去噪自动编码器实际上是由去噪自动编码器和多层感知器模型组成的。但是，由于堆叠去噪自动编码器的训练过程分为逐层预训练和整体调优两个阶段，所以对去噪自动编码器模型和多层感知器模型均需要进行相应修改。为了代码的完整性，我们这里列出了去噪自动编码器和多层感知器模型的相关代码，这些代码与前面章节讲述的内容有一定的重复，读者应该重点关注与前面章节所讲解的实现中不同的部分。

首先来看 MNIST 手写数字识别数据集载入，代码如下：

```
1 def load_datasets(self):
2     mnist = input_data.read_data_sets(self.datasets_dir,
3         one_hot=True)
4     X_train = mnist.train.images
5     y_train = mnist.train.labels
6     X_validation = mnist.validation.images
7     y_validation = mnist.validation.labels
8     X_test = mnist.test.images
9     y_test = mnist.test.labels
10    return X_train, y_train, X_validation, y_validation, \
11        X_test, y_test, mnist
```

第 2、3 行：调用 TensorFlow 的 `input_data` 的 `read_data_sets` 方法，第一个参数为数据集存放路径，第二个参数是标签集的格式。在原始 MNIST 数据集中，我们知道每个样本是 28×28 的黑白图片，对应的是 0~9 的数字标签，所以其格式为：[...784 (28×28) 像素点的值...][3]。其中，第一项为 784 (28×28) 个 0~1 的浮点数，0 代表黑色，1 代表白色；第二项的“3”代表这个样本是数字 3。为了后续处理方便，我们将标签[3]改为 one-hot 形式，因为标签代表 0~9 的数字，所以标签集为 10 维向量，每维上取值为 0 代表不是这个对应位置的数字，取值为 1 代表是这个对应位置的数字。其中，只有一维可以取 1，因此称之为 one-hot，还以上面的例子为例，标签集的格式就变为：[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]，因为第四位为 1，所以代表这个样本是数字 3。

第 4 行：取出训练样本集输入信号集 `X_train`，其为设计矩阵形式，每一行代表一个样本，行数为训练样本集中的样本数量。在这个例子中，就 `X_train` $\in \mathbb{R}^{55000 \times 784}$ ，其中训练样

本集中有 55000 个样本，每个样本是 784 (28×28) 维的图片。

第 5 行：取出训练样本集标签集 y_{train} ，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为训练样本集中的样本数量。在这个例子中，就 $y_{\text{train}} \in \mathbb{R}^{55000 \times 10}$ ，其中训练样本集中有 55000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 6 行：取出验证样本集输入信号集 $X_{\text{validation}}$ ，其为设计矩阵形式，每一行代表一个样本，行数为验证样本集中的样本数量。在这个例子中，就 $X_{\text{validation}} \in \mathbb{R}^{5000 \times 784}$ ，其中验证样本集中有 5000 个样本，每个样本是 784 (28×28) 维的图片。根据前面我们的讨论可以知道，在训练过程中，为了防止模型出现过拟合，模型的泛化能力降低（模型在训练样本集达到非常高的精度，但是在未见过的测试样本集或实际应用中，精度反而不高），通常会采用 Early Stopping 策略，就是在逻辑回归模型训练过程中，只用训练样本集对模型进行训练，每隔一定的时间间隔，计算模型在未见过的验证样本集上识别的精度，并记录迄今为止在验证样本集上取得的最高精度。我们会发现，在训练初期，验证样本集上的识别精度会稳步提高，但是到了一定阶段之后，验证样本集上的识别精度就不会再明显提高了，甚至开始逐渐下降，这就说明模型出现了过拟合，这时就可以停止模型训练，将在验证样本集上取得最佳识别精度的模型参数作为模型最终的参数。综上所述，验证样本集主要用于防止模型出现过拟合，为 Early Stopping 算法提供终止依据。

第 7 行：取出验证样本集标签集 $y_{\text{validation}}$ ，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为训练样本集中的样本数量。在这个例子中，就 $y_{\text{validation}} \in \mathbb{R}^{5000 \times 10}$ ，其中验证样本集中有 5000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 8 行：取出测试样本集输入信号集 X_{test} ，其为设计矩阵形式，每一行代表一个样本，行数为训练样本集中的样本数量。在这个例子中，就 $X_{\text{test}} \in \mathbb{R}^{10000 \times 784}$ ，其中训练样本集中有 10000 个样本，每个样本是 784 (28×28) 维的图片。测试样本集主要用于模型训练结束后对模型性能进行评估。由于模型没有见过测试样本集中的样本，可以模拟模型在实际部署之后的情况，模型在测试样本集上的识别精度，基本可以视为模型在实际应用中可以达到的精度。

第 9 行：取出测试样本集标签集 y_{test} ，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为测试样本集中的样本数量。在这个例子中，就 $y_{\text{test}} \in \mathbb{R}^{10000 \times 10}$ ，其中测试样本集中有 10000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 10、11 行：返回训练样本集输入信号集、训练样本集标签集、验证样本集输入信号集、验证样本集标签集、测试样本集输入信号集、测试样本集标签集。

再来看堆叠去噪自动编码器模型的训练方法，代码如下：

```
1 def train(self, mode=TRAIN_MODE_NEW, ckpt_file='work/dae.ckpt'):
2     X_train, y_train, X_validation, y_validation, \
3         X_test, y_test, mnist = self.load_datasets()
```



```

4     self.pretrain(X_train, X_validation)
5     if self.mlp_engine is None:
6         self.mlp_engine = Mlp_Engine(self.daes, 'datasets')
7     self.mlp_engine.train()

```

第 2、3 行：读入 MNIST 手写数字识别数据集，包括训练样本集输入样本集、训练样本集标签集、验证样本集输入样本集、验证样本集标签集、测试样本集输入样本集、测试样本集标签集。

第 4 行：调用预训练方法。我们知道堆叠去噪自动编码器训练包括两个阶段：第一阶段是逐层训练去噪自动编码器，第二阶段是将预训练好的去噪自动编码器堆叠在一起，形成一个标准的多层感知器模型，进行调优。本行代码就是调用逐层预训练去噪自动编码器。

第 5、6 行：如果没有初始化多层感知器模型，则初始化多层感知器模型。

第 7 行：调用多层感知器模型的训练方法，对参数进行调优。

接着来看堆叠去噪自动编码器的构造函数，代码如下：

```

1  def __init__(self):
2      self.datasets_dir = 'datasets/'
3      self.random_seed = 1
4      self.dae_W = []
5      self.dae_b = []
6      self.daes = []
7      self.dae_graphs = []
8      self.layers = [1024, 784, 512, 256]
9      self.name = 'sda'
10     prev = 784
11     self.mlp_engine = None
12     for idx, layer in enumerate(self.layers):
13         dae_str = 'dae_' + str(idx+1)
14         name = self.name + '_' + dae_str
15         tf_graph = tf.Graph()
16         self.daes.append(Dae_Engine(name, tf_graph=tf_graph, n=prev,
17                                   hidden_size=layer))
18         prev = layer
19     self.dae_graphs.append(tf_graph)

```

第 2 行：指定数据集存放位置。

第 3 行：随机数生成的种子。在网络权值和偏置值初始化时，都会用到随机数来进行初始化，如果每次生成的随机数均不相同，那么调试起来会非常不方便。因此，这里采用随机数种子，这样使用相同的种子，生成的随机数就是一样的，便于进行调试。

第 4 行：连接权值矩阵列表，每个预训练的去噪自动编码器对应一个元素。

第 5 行：偏置值列表，每个预训练的去噪自动编码器对应一个元素。

第 6 行：逐层预训练的去噪自动编码器列表。

第 7 行：逐层预训练的去噪自动编码器所对应的 Graph 列表，每个去噪自动编码器对应一个 Graph 对象。

第 8 行：定义堆叠去噪自动编码器的架构，第一个去噪自动编码器结构为 784—1024—784；第二个去噪自动编码器结构为 1024—784—1024；第三个去噪自动编码器结构为 784—512—784；第四个去噪自动编码器结构为 512—256—512。

第9行：定义模型名称。

第10行：定义输入层维度为784。

第11行：定义用于调优阶段的多层感知器模型。

第12行：利用第13~19行循环初始化堆叠去噪自动编码机的各个去噪自动编码器。

第13、14行：定义去噪自动编码器名称，需要多个去噪自动编码器的堆叠，这样可以帮助我们区分不同的去噪自动编码器。

第15行：建立一个新的 TensorFlow 计算图。

第16、17行：生成一个新的去噪自动编码器，并添加到去噪自动编码器列表中。

第18行：将本去噪自动编码器的隐藏层作为下一个去噪自动编码器的输入层。

第19行：将生成的 TensorFlow 的 Graph 添加到 Graph 列表中。

下面来看预训练过程，代码如下：

```
1 def pretrain(self, X_train, X_validation):
2     X_train_prev = X_train
3     X_validation_prev = X_validation
4     for idx, dae in enumerate(self.daes):
5         print('pretrain:{0}'.format(dae.name))
6         tf_graph = self.dae_graphs[idx]
7         X_train_prev, X_validation_prev = self.pretrain_dae(
8             self.dae_graphs[idx], dae,
9             X_train_prev, X_validation_prev)
10    return X_train_prev, X_validation_prev
```

第2、3行：每个去噪自动编码器的输入均为上一个去噪自动编码器隐藏层的内容，初始时将上一个去噪自动编码器隐藏层的值设置为训练样本集和验证样本集。

第4行：对所有去噪自动编码器列表中的去噪自动编码器循环第5~9行操作。

第6行：取出该去噪自动编码器对应的 TensorFlow 的 Graph。

第7~9行：预训练某个去噪自动编码器。

第10行：返回整个去噪自动编码机组的隐藏层状态。

下面来看每个去噪自动编码器的训练过程，代码如下：

```
1 def pretrain_dae(self, graph, dae, X_train, X_validation):
2     dae.train(X_train, X_validation)
3     X_train_next = dae.transform(graph, X_train)
4     X_validation_next = dae.transform(graph, X_validation)
5     return X_train_next, X_validation_next
```

第2行：调用去噪自动编码器的训练方法，进行预训练。

第3行：调用去噪自动编码器的 transform 方法，将训练样本集转变为其隐藏层输出的结果，用于对下一个去噪自动编码器的输入。

第4行：调用去噪自动编码器的 transform 方法，将验证样本集转变为其隐藏层输出的结果，用于对下一个去噪自动编码器的输入。

第5行：返回隐藏层的输出，以此作为下一个去噪自动编码器的输入。

以上就是堆叠去噪自动编码器预训练的全部过程，这里用到了与上一章有一定差别的去噪自动编码器模型，为了使读者有一个完整的认识，我们重新讲解这个去噪自动编码器

模型。

首先来看构造函数，代码如下：

```

1  def __init__(self, name='dae', tf_graph=tf.Graph(),
2      n=784, hidden_size=1024):
3      self.datasets_dir = 'datasets/'
4      self.name = name
5      self.random_seed = 1
6      self.input_data = None
7      self.input_labels = None
8      self.keep_prob = None
9      self.layer_nodes = []
10     self.train_step = None
11     self.cost = None
12     # TensorFlow objects
13     self.tf_graph = tf_graph
14     self.tf_session = None
15     self.tf_saver = None
16     self.tf_merged_summaries = None
17     self.tf_summary_writer = None
18     self.loss_func = 'cross_entropy'
19     self.enc_act_func = tf.nn.tanh
20     self.dec_act_func = tf.nn.tanh
21     self.num_epochs = 10
22     self.batch_size = 10
23     self.opt = 'adam'
24     self.learning_rate = 0.01
25     self.momentum = 0.9
26     self.corr_type = 'masking'
27     self.corr_frac = 0.1
28     self.regtype = 'l2'
29     self.regcoef = 5e-4
30     self.n = n
31     self.hidden_size = hidden_size

```

第 1、2 行：构造函数，如果用于堆叠去噪自动编码器中，需要指定名称，`graph` 为对应的 TensorFlow 的 `Graph`，输入层维度为 `n`，隐藏层维度为 `hidden_size`。

第 3 行：指定数据集存放目录。

第 4 行：定义并初始化名称属性。

第 5 行：因为在网络权值和偏置值初始化时，都会用到随机数来进行初始化，如果每次生成的随机数均不相同，那么调试起来会非常不方便，因此这里采用随机数种子，这样使用相同的种子，生成的随机数就是一样的，便于进行调试。

第 6 行：定义输入数据属性。

第 7 行：定义输入信号标签。

第 8 行：定义当采用 Dropout 调整技术时，神经元保持活跃的比例。

第 9 行：组成神经网络每一层的序列。

第 10 行：训练方法的 TensorFlow 计算图定义。

第 11 行：代价函数定义。

第 13 行：定义并初始化 TensorFlow 的 `Graph` 属性。

第 14 行：保存用于运行的 TensorFlow 的会话对象。

第 15 行：用于保存和恢复模型参数的 TensorFlow 的 saver 对象。

第 16、17 行：用于统计分析的属性。

第 18 行：定义原始代价函数为交叉熵函数，最终的代价函数为原始代价函数再加上调整项，如 L2 权值衰减等。

第 19 行：指定用于编码的隐藏层神经元激活函数。

第 20 行：指定用于解码的输出层神经元激活函数。

第 21 行：指定完整训练完整个训练样本集的遍数。

第 22 行：迷你批次的大小。

第 23 行：指定优化算法。

第 24 行：指定学习率超参数。

第 25 行：指定动量项超参数。

第 26 行：指定噪声形式为掩码失活形式。

第 27 行：加入噪声的比例为 10%，即有 10% 的神经元的输出将被随机设置为 0。

第 28 行：调整项为 L2（权值衰减）。

第 29 行：L2 调整项的系数。

第 30 行：定义输入向量维度 n。

第 31 行：定义隐藏层神经元数。

下面来看看去噪自动编码器模型的创建，代码如下：

```
1 def build_model(self):
2     print('Build Denoising Autoencoder Model v0.0.8')
3     print('begin to build the model')
4     self.X = tf.placeholder(shape=[None, self.n], dtype=tf.float32)
5     self.y = tf.placeholder(shape=[None, self.n], dtype=tf.float32)
6     self.keep_prob = tf.placeholder(dtype=tf.float32, name='keep_prob')
7     self.W1 = tf.Variable(
8         tf.truncated_normal(
9             shape=[self.n, self.hidden_size], mean=0.0, stddev=0.1),
10        name='W1')
11     self.b2 = tf.Variable(tf.constant(
12         0.001, shape=[self.hidden_size]), name='b2')
13     self.b3 = tf.Variable(tf.constant(
14         0.001, shape=[self.n]), name='b3')
15     with tf.name_scope('encoder'):
16         z2 = tf.matmul(self.X, self.W1) + self.b2
17         self.a2 = tf.nn.tanh(z2)
18     with tf.name_scope('decoder'):
19         z3 = tf.matmul(self.a2, tf.transpose(self.W1)) + self.b3
20         a3 = tf.nn.tanh(z3)
21     self.y_ = a3
22     r_y_ = tf.clip_by_value(self.y_, 1e-10, float('inf'))
23     r_l_y_ = tf.clip_by_value(1 - self.y_, 1e-10, float('inf'))
24     cost = - tf.reduce_mean(tf.add(
25         tf.multiply(self.y, tf.log(r_y_)),
26         tf.multiply(tf.subtract(1.0, self.y), tf.log(r_l_y_))))
27     self.J = cost + self.regcoef * tf.nn.l2_loss([self.W1])
```

```

28 self.train_op = tf.train.AdamOptimizer(0.001,0.9,0.9,1e-08).\
29     minimize(self.J)

```

第 4 行：定义网络输入信号的 `placeholder`，以第一个去噪自动编码器为例，其为 784 维。在训练过程中，我们输入到网络的为加入噪声后的图像；而在运行时，输入网络的则为原始图像。

第 5 行：定义网络输出层输出信号，其为 784 维，我们的目标就是使 y 和 x 尽量相等。输出层仅在网络预训练时会用到，当预训练结束后，将去掉输出层，直接将隐藏层作为下一个去噪自动编码器的输入层。

第 6 行：定义采用 Dropout 调整项时，神经元的活跃概率。

第 7~10 行：定义输入层到隐藏层的连接权值对象，维度为 784×1024 ，用均值为 0、标准差为 0.1 的正态分布随机数进行初始化。

第 11、12 行：定义隐藏层偏置值，采用均值为 0、标准差为 0.001 的正态分布随机数进行初始化。

第 13、14 行：定义输出层偏置值，采用均值为 0、标准差为 0.001 的正态分布随机数进行初始化。注意：由于隐藏层到输出层的连接权值矩阵是输出层到隐藏层连接权值矩阵的转换，所以不需要定义隐藏层到输出层的连接权值矩阵。

第 15~17 行：定义输入层到隐藏层的编码器部分。首先将输入信号与连接权值矩阵相乘，再加上隐藏层偏置值，最后经过双曲正切激活函数，求出隐藏层输出。读者可以参照前面章节的内容加入 Dropout 调整项，再求出输出，这里我们没有给出代码，读者可以将其当做一个练习，加入适当的 Dropout，尝试重新训练网络。

第 18~21 行：定义隐藏层到输出层的解码器部分，将隐藏层输出信号与输入层到隐藏层连接权值矩阵 $W1$ 的转置相乘，加上输出层的偏置值，经过双曲正切激活函数，求出输出层的输出信号 y_- 。

第 22、23 行：因为用计算机表示浮点数有精度问题，所以对 y_- 和 $1-y_-$ ，如果值小于 $1e-10$ ，则取值为 $1e-10$ 。

第 24~26 行：定义代价函数，假设我们有 m 个训练样本，则计算公式为：

$$\text{cost} = y \log \hat{y} + (1 - y) \log(1 - \hat{y})$$

第 27 行：定义最终的代价函数为加上 L2 调整项（权值衰减）后的值，其中 `tf.nn.l2_loss` 函数用于计算权值衰减项，这里只对连接权值进行调整。`self.regcoef` 为连接权值衰减项的系数。

第 28、29 行：定义训练操作，采用 Adam 优化算法，求代价函数为最小值时参数的值。下面来看网络的训练方法，如下所示：

```

1 def train(self, X_train, X_validation, mode=TRAIN_MODE_NEW):
2     ckpt_file='work/{0}.ckpt'.format(self.name)
3     with self.tf_graph.as_default():
4         self.build_model()
5         saver = tf.train.Saver()
6         with tf.Session() as sess:
7             sess.run(tf.global_variables_initializer())
8             for epoch in range(self.num_epochs):

```

```

9         X_train_prime = self.add_noise(sess, X_train,
10                                         self.corr_frac)
11         shuff = list(zip(X_train, X_train_prime))
12         np.random.shuffle(shuff)
13         batches = [_ for _ in self.gen_mini_batches(shuff,
14                                                     self.batch_size)]
15         batch_idx = 1
16         for batch in batches:
17             X_batch_raw, X_prime_batch_raw = zip(*batch)
18             X_batch = np.array(X_batch_raw).astype(np.float32)
19             X_prime_batch = np.array(X_prime_batch_raw).\
20                                     astype(np.float32)
21             batch_idx += 1
22             opv, loss = sess.run([self.train_op, self.J],
23                                 feed_dict={self.X: X_prime_batch,
24                                             self.y: X_batch})
25             if batch_idx % 1000 == 0:
26                 print('epoch{0}_batch{1}: {2}'.format(epoch,
27                                                         batch_idx, loss))
28             saver.save(sess, ckpt_file)

```

第 2 行：定义模型的保存文件，每个去噪自动编码器的模型文件要不同，否则就会互相覆盖了。

第 3 行：启动 TensorFlow 的 graph。

第 4 行：调用 `build_model` 创建去噪自动编码器模型。

第 5 行：定义 `tf.train.Saver` 对象，用于保存训练阶段的参数值等信息。

第 6 行：启动 TensorFlow 会话。

第 7 行：初始化全局变量。

第 8 行：对每个全量训练样本集训练，执行第 9~27 行操作。

第 9、10 行：向训练样本集输入样本集添加 `self.corr_frac` 比例的 Masking 噪声。就是这个模型被称为去噪自动编码器模型的原因，网络的任务除了恢复输入信号，还需要去掉这些人为加上去的噪声。

第 11 行：把原始训练样本集输入样本集和加入噪声的训练样本集输入样本集，以样本一一对应的方式形成一个列表。

第 12 行：对上面形成的列表进行随机洗牌。

第 13、14 行：以 `batch_size` 为单位，将样本集拆分为迷你批次。

第 15 行：记录迷你批次数号。

第 16 行：对于每个迷你批次，循环第 17~27 行操作。

第 17 行：从批次中取出原始样本集和加入噪声的样本集。

第 18 行：将原始样本集变为 `numpy.ndarray` 类型。

第 19、20 行：将加入噪声的样本集变为 `numpy.ndarray` 类型。

第 21 行：求出迷你批次数号。

第 22~24 行：通过 TensorFlow 求出代价函数值。注意：此时 `self.X` 的输入为加入噪声的样本集，`self.y` 输入的是原始样本集。

第 25~27 行：每隔 100 次打印一次训练信息。

第 28 行：保存网络参数。

由于我们的网络比较小，所以运行训练方法时只需要几分钟就可以完成预训练过程。在预训练过程中，还用到了两个方法：向输入样本集加入 **masking** 噪声和产生迷你批次。向输入样本集加入 **masking** 噪声的代码如下：

```
1 def add_noise(self, sess, X, corr_frac):
2     X_prime = X.copy()
3     rand = tf.random_uniform(X.shape)
4     X_prime[sess.run(tf.nn.relu(tf.sign(corr_frac - rand))).astype(np.bool)] = 0
5     return X_prime
```

第 2 行：复制一份输入样本集。

第 3 行：利用均匀分布产生 0~1 的随机数。

第 4 行：将产生的随机数与噪声比例相减，取其符号，经过 **ReLU** 函数，如果为负数时值为 0，如果为正数其值不变，也就是说，有 **corr_frac** 比例的输入信号将被置 0。

第 5 行：返回加入噪声的样本集。

产生迷你批次的代码如下：

```
1 def gen_mini_batches(self, X, batch_size):
2     X = np.array(X)
3     for i in range(0, X.shape[0], batch_size):
4         yield X[i:i + batch_size]
```

这段代码先将样本变为 **numpy** 的数组，将其分割为 **batch_size** 大小的子数组，再用 **yield** 函数将其作为一个序列。

与上一章中的去噪自动编码器模型最大的不同是，本章中的去噪自动编码器需要将隐藏层接入下一个去噪自动编码器，因此需要将输入信号转换成隐藏层的输出信号，并以其作为下一个去噪自动编码器的输入信号，所以需要将输入信号转换为隐藏层的输出信号，代码如下：

```
1 def transform(self, graph, data):
2     ckpt_file='work/{0}.ckpt'.format(self.name)
3     with self.tf_graph.as_default():
4         saver = tf.train.Saver()
5         with tf.Session() as sess:
6             saver.restore(sess, ckpt_file)
7             feed = {self.X: data, self.keep_prob: 1}
8             return sess.run(self.a2, feed_dict=feed)
```

第 2 行：指定模型参数的保存文件，模型参数文件以自己的名称为前缀，因此不同去噪自动编码器之间不会重复。

第 3 行：打开本去噪自动编码器对应的 **TensorFlow** 计算图。

第 4 行：创建 **TensorFlow** 的 **saver** 对象，用于恢复模型的参数和超参数。

第 5 行：启动 **TensorFlow** 会话。

第 6 行：从模型参数文件中恢复模型的参数和超参数。

第 7 行：将原始输入数据输入到网络中，**Dropout** 时神经元激活率为 100%（不采用 **Dropout** 技术）。

第8行：求出隐藏层的输出并返回。

当预训练完所有的去噪自动编码器之后，就进入了整体网络调优阶段，可以把这个阶段假想为在已经预训练好的去噪自动编码器之上叠加一个用于分类的 Softmax 回归层，形成一个完整的多层感知器模型，相当于已经知道了多层感知器模型中的连接权值矩阵和偏置值，在此基础上继续训练多层感知器模型。

首先来看多层感知器模型的构造函数，代码如下：

```
1 def __init__(self, daes, datasets_dir):
2     self.datasets_dir = datasets_dir
3     self.batch_size = 100
4     self.n = 784
5     self.k = 10
6     self.L = np.array([self.n, 1024, 784, 512, 256, self.k])
7     self.lanmeda = 0.001
8     self.keep_prob_val = 0.75
9     self.daes = daes
10    self.model = {}
```

第2行：指定数据集文件存放目录。

第3行：指定迷你批次大小。

第4行：指定输入向量维度为 784。

第5行：输出信号类别为 10，即 0~9 这 10 个数字。

第6行：定义网络结构的第一层为 784 维，第二层为 1024 维，第三层为 784 维，第四层为 512 维，第五层为 256 维，第六层为 10 维，对应 0~9 这 10 个数字。

第7行：L2 调整项（权值衰减）的系数。

第8行：定义采用 Dropout 调整技术时，神经元激活的概率，在本例中 75% 的神经元保持输出不变。

第9行：引入预训练好的去噪自动编码器，主要用于读取已经训练好的参数和超参数。

第10行：定义保存模型变量的属性。

接下来看模型构建过程，代码如下：

```
1 def build_model(self, mode='train'):
2     print('mode={0}'.format(mode))
3     self.X = tf.placeholder(tf.float32, [None, 784])
4     self.y = tf.placeholder(tf.float32, [None, 10])
5     self.keep_prob = tf.placeholder(tf.float32) #Dropout 失活率
6     if 'train' == mode:
7         # 取出预训练去噪自动编码器参数
8         with self.daes[0].tf_graph.as_default():
9             with tf.Session() as sess:
10                sess.run(tf.global_variables_initializer())
11                dae0_W1 = sess.run(self.daes[0].W1)
12                dae0_b2 = sess.run(self.daes[0].b2)
13            with self.daes[1].tf_graph.as_default():
14                with tf.Session() as sess:
15                    sess.run(tf.global_variables_initializer())
16                    dae1_W1 = sess.run(self.daes[1].W1)
17                    dae1_b2 = sess.run(self.daes[1].b2)
18            with self.daes[2].tf_graph.as_default():
```



```

19         with tf.Session() as sess:
20             sess.run(tf.global_variables_initializer())
21             dae2_W1 = sess.run(self.daes[2].W1)
22             dae2_b2 = sess.run(self.daes[2].b2)
23         with self.daes[3].tf_graph.as_default():
24             with tf.Session() as sess:
25                 sess.run(tf.global_variables_initializer())
26                 dae3_W1 = sess.run(self.daes[3].W1)
27                 dae3_b2 = sess.run(self.daes[3].b2)
28     # 从 784 维到 1024 维的去噪自动编码器
29     if 'train' == mode:
30         self.W_1 = tf.Variable(dae0_W1, name='W_1')
31         self.b_2 = tf.Variable(dae0_b2, name='b_2')
32     else:
33         self.W_1 = tf.Variable(tf.truncated_normal([784, 1024], mean=0.0,
34             stddev=0.1), name='W_1')
35         self.b_2 = tf.Variable(tf.zeros([1024]), name='b_2')
36     self.z_2 = tf.matmul(self.X, self.W_1) + self.b_2
37     self.a_2 = tf.nn.tanh(self.z_2) # tf.nn.relu(self.z_2)
38     self.a_2_dropout = tf.nn.dropout(self.a_2, self.keep_prob)
39     # 从 1024 维到 784 维的去噪自动编码器
40     if 'train' == mode:
41         self.W_2 = tf.Variable(dae1_W1, name='W_2')
42         self.b_3 = tf.Variable(dae1_b2, name='b_3')
43     else:
44         self.W_2 = tf.Variable(tf.truncated_normal([1024, 784], mean=0.0,
45             stddev=0.1), name='W_2')
46         self.b_3 = tf.Variable(tf.zeros([784]), name='b_3')
47     self.z_3 = tf.matmul(self.a_2_dropout, self.W_2) + self.b_3
48     self.a_3 = tf.nn.tanh(self.z_3) # tf.nn.relu(self.z_3)
49     self.a_3_dropout = tf.nn.dropout(self.a_3, self.keep_prob)
50     # 从 784 维到 512 维的去噪自动编码器
51     if 'train' == mode:
52         self.W_3 = tf.Variable(dae2_W1, name='W_3')
53         self.b_4 = tf.Variable(dae2_b2, name='b_4')
54     else:
55         self.W_3 = tf.Variable(tf.truncated_normal([784, 512], mean=0.0,
56             stddev=0.1), name='W_3')
57         self.b_4 = tf.Variable(tf.zeros([512]), name='b_4')
58     self.z_4 = tf.matmul(self.a_3_dropout, self.W_3) + self.b_4
59     self.a_4 = tf.nn.tanh(self.z_4) # tf.nn.relu(self.z_4)
60     self.a_4_dropout = tf.nn.dropout(self.a_4, self.keep_prob)
61     # 从 512 维到 256 维的去噪自动编码器
62     if 'train' == mode:
63         self.W_4 = tf.Variable(dae3_W1, name='W_4')
64         self.b_5 = tf.Variable(dae3_b2, name='b_5')
65     else:
66         self.W_4 = tf.Variable(tf.truncated_normal([512, 256], mean=0.0,
67             stddev=0.1), name='W_4')
68         self.b_5 = tf.Variable(tf.zeros([256]), name='b_5')
69     self.z_5 = tf.matmul(self.a_4_dropout, self.W_4) + self.b_5
70     self.a_5 = tf.nn.tanh(self.z_5) # tf.nn.relu(self.z_5)
71     self.a_5_dropout = tf.nn.dropout(self.a_5, self.keep_prob)
72     # 输出层
73     self.W_5 = tf.Variable(tf.zeros([256, 10]))
74     self.b_6 = tf.Variable(tf.zeros([10]))

```

```

75 self.z_6 = tf.matmul(self.a_5_dropout, self.W_5) + self.b_6
76 self.y_ = tf.nn.softmax(self.z_6)
77 #训练部分
78 self.cross_entropy = tf.reduce_mean(-tf.reduce_sum(
79     self.y * tf.log(self.y_),
80     reduction_indices=[1]))
81 #train_step = tf.train.AdagradOptimizer(0.3).minimize(cross_entropy)
82 self.loss = self.cross_entropy + self.lanmeda*(
83     tf.reduce_sum(self.W_1**2) +
84     tf.reduce_sum(self.W_2**2) + tf.reduce_sum(self.W_3**2) +
85     tf.reduce_sum(self.W_4**2) + tf.reduce_sum(self.W_5**2))
86 self.train_step = tf.train.AdamOptimizer(learning_rate=0.001, beta1=0.9,
87     beta2=0.999, epsilon=1e-08, use_locking=False,
88     name='Adam').minimize(self.loss)
89 self.correct_prediction = tf.equal(tf.argmax(self.y_, 1),
90     tf.argmax(self.y, 1))
91 self.accuracy = tf.reduce_mean(tf.cast(self.correct_prediction,
92     tf.float32))
93 return self.X, self.y_, self.y, self.keep_prob, self.cross_entropy, \
94     self.train_step, self.correct_prediction, self.accuracy

```

第3行：定义输入信号的 placeholder。

第4行：定义正确标签结果的 placeholder。

第5行：采用 Dropout 调整技术时，神经元处于激活状态的比例。

第6行：如果创建训练时的模型，需要从已经预训练好的去噪自动编码器中读取参数和超参数；如果创建运行状态时的模型，则无须这一步。

第8行：打开第一个去噪自动编码器对应的 TensorFlow 的计算图。

第9行：开启 TensorFlow 会话。

第10行：初始化全局变量。

第11行：取出第一个去噪自动编码机的连接权值矩阵。

第12行：取出第一个去噪自动编码机的偏置值向量。

第13行：打开第二个去噪自动编码机的 TensorFlow 计算图。

第14行：开启 TensorFlow 会话。

第15行：初始化全局变量。

第16行：取出第二个去噪自动编码机的连接权值矩阵。

第17行：取出第二个去噪自动编码机的偏置值向量。

第18行：打开第三个去噪自动编码机的 TensorFlow 计算图。

第19行：打开 TensorFlow 会话。

第20行：初始化全局变量。

第21行：取出第三个去噪自动编码机的连接权值矩阵。

第22行：取出第三个去噪自动编码机的偏置值向量。

第23行：打开第四个去噪自动编码机的 TensorFlow 计算图。

第24行：开启 TensorFlow 会话。

第25行：初始化全局变量。

第26行：取出第四个去噪自动编码机的连接权值矩阵。

第 27 行：取出第四个去噪自动编码器的偏置值向量。

第 29~31 行：如果创建训练模型，则将第 1 层到第 2 层的连接权值初始化为第 1 个去噪自动编码器的连接权值，第 2 层的偏置值为第 1 个去噪自动编码器的偏置值。

第 32~35 行：如果创建运行状态的模型，则用均值为 0、标准差为 0.1 的正态分布随机数初始化第 1 层到第 2 层的连接权值矩阵，用 0 初始化第 2 层的偏置值。

第 36 行：求出第 2 层的输入值。

第 37 行：利用双曲正切函数求出第 2 层的原始输出，因为我们的去噪自动编码器采用的是双曲正切激活函数。

第 38 行：采用 Dropout 调整技术，随机选择 `self.keep_prob` 比例的神经元处于激活状态，其余神经元的输出值为 0，得到最终第 2 层的输出值。

第 40~42 行：如果创建训练模型，则将第 2 层到第 3 层的连接权值初始化为第 2 个去噪自动编码器的连接权值，第 3 层的偏置值为第 2 个去噪自动编码器的偏置值。

第 43~46 行：如果创建运行状态的模型，则用均值为 0、标准差为 0.1 的正态分布随机数初始化第 2 层到第 3 层的连接权值矩阵，用 0 初始化第 3 层的偏置值。

第 47 行：求出第 3 层的输入值。

第 48 行：利用双曲正切函数求出第 3 层的原始输出，因为我们的去噪自动编码器采用的是双曲正切激活函数。

第 49 行：采用 Dropout 调整技术，随机选择 `self.keep_prob` 比例的神经元处于激活状态，其余神经元的输出值为 0，得到最终第 3 层的输出值。

第 51~53 行：如果创建训练模型，则将第 3 层到第 4 层的连接权值初始化为第 3 个去噪自动编码器的连接权值，第 4 层的偏置值为第 3 个去噪自动编码器的偏置值。

第 54~57 行：如果创建运行状态的模型，则用均值为 0、标准差为 0.1 的正态分布随机数初始化第 3 层到第 4 层的连接权值矩阵，用 0 初始化第 4 层的偏置值。

第 58 行：求出第 4 层的输入值。

第 59 行：利用双曲正切函数求出第 4 层的原始输出，因为我们的去噪自动编码器采用的是双曲正切激活函数。

第 60 行：采用 Dropout 调整技术，随机选择 `self.keep_prob` 比例的神经元处于激活状态，其余神经元的输出值为 0，得到最终第 4 层的输出值。

第 62~64 行：如果创建训练模型，则将第 4 层到第 5 层的连接权值初始化为第 4 个去噪自动编码器的连接权值，第 5 层的偏置值为第 4 个去噪自动编码器的偏置值。

第 65~68 行：如果创建运行状态的模型，则用均值为 0、标准差为 0.1 的正态分布随机数初始化第 4 层到第 5 层的连接权值矩阵，用 0 初始化第 5 层的偏置值。

第 69 行：求出第 5 层的输入值。

第 70 行：利用双曲正切函数求出第 5 层的原始输出，因为我们的去噪自动编码器采用的是双曲正切激活函数。

第 71 行：采用 Dropout 调整技术，随机选择 `self.keep_prob` 比例的神经元处于激活状态，其余神经元的输出值为 0，得到最终第 5 层的输出值。

第 73 行：定义第 5 层到第 6 层（输出层）的连接权值矩阵，并用全 0 来进行初始化。

第 74 行：定义第 6 层（输出层）的偏置值，并用全 0 来进行初始化。

第 75 行：求出第 6 层的输入值。

第 77 行：利用 Softmax 函数求出第 6 层的输出，即计算出的标签分类结果。

第 78~80 行：定义交叉熵 cross_entropy 的计算方法。

第 81 行：定义调整项 L2 权值衰减系数 λ 。

第 82~85 行：定义最终代价函数，值为交叉熵再加上 L2 调整项（权值衰减）。

第 86~88 行：采用梯度下降算法和 Adam 优化算法，利用优化算法求使代价函数达到最小值时连接权值 W 和偏移量 b 的值。

第 89、90 行：利用 TensorFlow 的 argmax 函数，分别求出计算类别向量每个样本的最大值下标和类别向量每个样本的最大值下标，并对其进行比较。

第 91、92 行：求出预测精度，首先调用 TensorFlow 的 cast 函数，将第 16 行的结果变为浮点数列表：[1.0, 1.0, 0.0, 1.0, 1.0]，我们这里假设只有 5 个样本。再调用 TensorFlow 的 reduce_mean 函数求出这个列表的平均值：(1.0+1.0+0.0+1.0+1.0)/5=0.8，这个值就是模型预测的精度。

第 93、94 行：返回模型定义的 X, W, b, y_, y, cross_entropy, train_step, correct_prediction, accuracy。

接下来是训练方法，代码如下：

```
1 def train(self, mode=TRAIN_MODE_NEW, ckpt_file='work/lgr.ckpt'):
2     X_train, y_train, X_validation, y_validation, X_test, \
3         y_test, mnist = self.load_datasets()
4     X, y_, y, keep_prob, cross_entropy, train_step, correct_prediction, \
5         accuracy = self.build_model()
6     epochs = 10
7     saver = tf.train.Saver()
8     total_batch = int(mnist.train.num_examples/self.batch_size)
9     check_interval = 50
10    best_accuracy = -0.01
11    improve_threthold = 1.005
12    no_improve_steps = 0
13    max_no_improve_steps = 3000
14    is_early_stop = False
15    eval_runs = 0
16    eval_times = []
17    train_accs = []
18    validation_accs = []
19    with tf.Session() as sess:
20        sess.run(tf.global_variables_initializer())
21        if Mlp_Engine.TRAIN_MODE_CONTINUE == mode:
22            saver.restore(sess, ckpt_file)
23        for epoch in range(epochs):
24            if is_early_stop:
25                break
26            for batch_idx in range(total_batch):
27                if no_improve_steps >= max_no_improve_steps:
28                    is_early_stop = True
29                    break
30            X_mb, y_mb = mnist.train.next_batch(self.batch_size)
```

```

31         sess.run(train_step, feed_dict={X: X_mb, y: y_mb,
32             keep_prob: self.keep_prob_val})
33     no_improve_steps += 1
34     if batch_idx % check_interval == 0:
35         eval_runs += 1
36         eval_times.append(eval_runs)
37         train_accuracy = sess.run(accuracy,
38             feed_dict={X: X_train, y: y_train, keep_prob: 1.0})
39         train_accs.append(train_accuracy)
40         validation_accuracy = sess.run(accuracy,
41             feed_dict={X: X_validation, y: y_validation,
42                 keep_prob: 1.0})
43         validation_accs.append(validation_accuracy)
44         if best_accuracy < validation_accuracy:
45             if validation_accuracy / best_accuracy >= \
46                 improve_threthold:
47                 no_improve_steps = 0
48                 best_accuracy = validation_accuracy
49                 saver.save(sess, ckpt_file)
50                 print('{0}:{1}# train:{2}, validation:{3}'.format(
51                     epoch, batch_idx, train_accuracy,
52                     validation_accuracy))
53         print(sess.run(accuracy, feed_dict={X: X_test,
54             y: y_test, keep_prob: 1.0}))
55     plt.figure(1)
56     plt.subplot(111)
57     plt.plot(eval_times, train_accs, 'b-', label='train accuracy')
58     plt.plot(eval_times, validation_accs, 'r-',
59         label='validation accuracy')
60     plt.title('accuracy trend')
61     plt.legend(loc='lower right')
62     plt.show()

```

第 2、3 行：读入训练样本集输入信号集、训练样本集标签集、验证样本集输入信号集、验证样本集标签集、测试样本集输入信号集、测试样本集标签集。

第 4、5 行：创建模型，这里创建的模型是 **ReLU** 神经元模型。

第 6 行：循环学习整个训练样本集遍数。

第 7 行：初始化 TensorFlow 模型保存和恢复对象 **saver**。

第 8 行：用用户训练样本集中样本数除以迷你批次大小，得到迷你批次数量 **total_batch**。

第 9 行：每隔 50 次迷你批次学习，计算在验证样本集上的精度。

第 10 行：保存在验证样本集上所取得的最好的验证样本集精度。

第 11 行：定义验证样本集上精度提高 0.5% 时才算显著提高。

第 12 行：记录在验证样本集上精度没有显著提高学习迷你批次的次数。

第 13 行：在验证样本集精度最大没有显著提高的情况下，允许学习迷你批次的次数。

第 14 行：是否需终止学习过程。

第 15 行：评估验证样本集上识别精度的次数。

第 16 行：用 **eval_times** 列表来保存评估次数，作为后面绘制识别精度趋势图的横坐标。

第 17 行：用 **train_accs** 列表保存在训练样本集上每次评估时的识别精度，作为后面图形深色曲线的纵坐标。

第 18 行：用 **validation_accs** 列表保存在验证样本集上每次评估时的识别精度，作为后

面图形浅色曲线的纵坐标。

第 19 行：启动 TensorFlow 会话。

第 20 行：初始化全局参数。

第 21、22 行：如果模式为 TRAIN_MODE_CONTINUE，则读入以前保存的 ckpt 模型文件，初始化模型参数。

第 23 行：循环第 20~52 行操作，对整个训练样本集进行一次学习。

第 24、25 行：如果 is_early_stop 为真，则终止本层循环。

第 26 行：循环第 23~52 行操作，对一个迷你批次进行学习。

第 27~29 行：如果验证样本集上识别精度没有显著改善的迷你批次学习次数大于最大允许的验证样本集上识别精度没有显著改善的迷你批次学习次数，则将 is_early_stop 置为真，并退出本层循环。这会直接触发第 24、25 行操作，终止外层循环，代表学习过程结束。

第 30 行：从训练样本集中取出一个迷你批次的输入信号集 X_{mb} 和标签集 y_{mb} 。

第 31、32 行：调用 TensorFlow 计算模型输出、代价函数，求出代价函数对参数的导数，并应用梯度下降算法更新模型参数值。注意，此时我们采用 Dropout 调整技术，将隐藏层神经元保留比例作为一个参数 keep_prob 传给模型。

第 33 行：将验证样本集没有显著改善的迷你批次学习次数加 1。

第 34 行：如果连续进行了指定次数的迷你批次学习，则计算统计信息。

第 35 行：识别精度评估次数加 1。

第 36 行：将识别精度评估次数加入 eval_times（图形横坐标）列表中。

第 37、38 行：计算训练样本集上的识别精度。

第 39 行：将训练样本集上的识别精度加入训练样本集识别精度列表 train_accs 中。

第 40~42 行：计算验证样本集上的识别精度。注意，此时我们采用 dropout 调整技术，将隐藏层神经元保留比例作为一个参数 keep_prob 传给模型，这里我们给出的是 1.0，即全部保留。这是一个惯例，即在训练时使用 Dropout 调整技术，在评估和运行时不使用 Dropout 调整技术，读者一定要注意。

第 43 行：将验证样本集上的识别精度加入到验证样本集识别精度列表 validation_accs 中。

第 44 行：如果验证样本集上最佳识别精度小于当前的验证样本集上的识别精度，执行第 45~49 行操作。

第 45~47 行：如果当前验证样本集上的识别精度比之前的最佳识别精度提高 0.5% 以上，则将验证样本集没有显著改善的迷你批次学习次数设为 0。

第 48 行：将验证样本集上最佳识别精度的值设置为当前验证样本集上识别精度的值。

第 49 行：将当前模型参数保存到 ckpt 模型文件中。

第 50~52 行：打印训练状态信息。

第 53、54 行：训练完成后，计算测试样本集上的识别精度，并打印出来。

第 55、56 行：初始化 matplotlib 绘图库。

第 57 行：绘制训练样本集上识别精度的变化趋势曲线，用深色绘制。

第 58、59 行：绘制验证样本集上识别精度的变化趋势曲线，用浅色绘制。

第 60 行：设置图形标题。
第 61 行：在右下角添加图例。
第 62 行：具体绘制图像。
运行堆叠去噪自动编码器的训练方法，会有如图 9.1～图 9.3 所示的结果输出。

```
pretrain:sda_dae_2
Build Denoising Autoencoder Model v0.0.8
begin to build the model
epoch0_batch1000: -4.874337196350098
epoch0_batch2000: -5.687304496765137
epoch0_batch3000: -6.3144965171813965
epoch0_batch4000: -6.3294243812561035
epoch0_batch5000: -6.167352676391602
epoch1_batch1000: -7.508970260620117
epoch1_batch2000: -6.271933078765869
epoch1_batch3000: -6.850400924682617
epoch1_batch4000: -6.0743842124938965
epoch1_batch5000: -6.163639545440674
epoch2_batch1000: -6.085707664489746
epoch2_batch2000: -6.485960006713867
epoch2_batch3000: -6.261335849761963
epoch2_batch4000: -5.888795852661133
epoch2_batch5000: -6.546745777130127
```

图 9.1 预训练阶段结果输出

```
0:0# train:0.30058181285858154, validation:0.2996000051498413
0:50# train:0.8688727021217346, validation:0.8766000270843506
0:100# train:0.9099454283714294, validation:0.9168000221252441
0:150# train:0.8964545726776123, validation:0.909600019454956
0:200# train:0.9191272854804993, validation:0.9233999848365784
0:250# train:0.8945454359054565, validation:0.900600016117096
0:300# train:0.9241999983787537, validation:0.9251999855041504
0:350# train:0.9077090620994568, validation:0.9168000221252441
0:400# train:0.9197636246681213, validation:0.9254000186920166
0:450# train:0.9108909368515015, validation:0.9151999950408936
0:500# train:0.9336000084877014, validation:0.9383999705314636
1:0# train:0.9269818067550659, validation:0.9344000220298767
1:50# train:0.9176181554794312, validation:0.9287999868392944
1:100# train:0.9156000018119812, validation:0.921999990940094
```

图 9.2 调优阶段结果输出

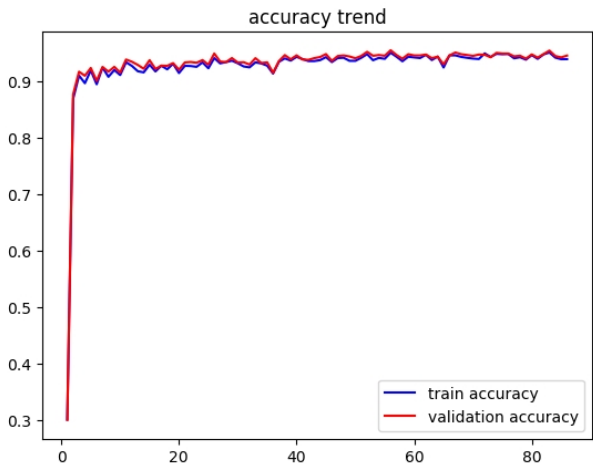


图 9.3 调优阶段训练样本集和验证样本集上的识别精度

下面是运行方法，在堆叠去噪自动编码器中，训练阶段有预训练和调优两个阶段。但

是在运行阶段，其就是一个标准的多层感知器模型，与去噪自动编码器没有任何关系，代码如下：

```

1 def run(self, ckpt_file='work/lgr.ckpt'):
2     print('run.....')
3     img_file = 'datasets/test5.png'
4     img = io.imread(img_file, as_grey=True)
5     raw = [1 if x<0.5 else 0 for x in img.reshape(784)]
6     #sample = np.array(raw)
7     X_train, y_train, X_validation, y_validation, \
8         X_test, y_test, mnist = self.load_datasets()
9     sample = X_test[102]
10    X_run = sample.reshape(1, 784)
11    digit = -1
12    with tf.Graph().as_default():
13        X, y_, y, keep_prob, cross_entropy, train_step, correct_prediction, \
14            accuracy = self.build_model(mode='run')
15        saver = tf.train.Saver()
16        with tf.Session() as sess:
17            sess.run(tf.global_variables_initializer())
18            saver.restore(sess, ckpt_file)
19            rst = sess.run(y_, feed_dict={X: X_run, keep_prob: 1.0})
20            print('rst:{0}'.format(rst))
21            max_prob = -0.1
22            for idx in range(10):
23                if max_prob < rst[0][idx]:
24                    max_prob = rst[0][idx]
25                    digit = idx
26            # W_1_1
27            W_1 = sess.run(self.W_1)
28            wight_map = W_1[:,0].reshape(28, 28)
29            a_2 = sess.run(self.a_2, feed_dict={X: X_run, \
30                keep_prob: 1.0})
31            a_2_raw = a_2[0]
32            a_2_img = a_2_raw[0:784]
33            feature_map = a_2_img.reshape(28, 28)
34            img_in = sample.reshape(28, 28)
35            plt.figure(1)
36            plt.subplot(131)
37            plt.imshow(img_in, cmap='gray')
38            plt.title('result:{0}'.format(digit))
39            plt.axis('off')
40            plt.subplot(132)
41            plt.imshow(wight_map, cmap='gray')
42            plt.axis('off')
43            plt.title('wight row')
44            plt.subplot(133)
45            plt.imshow(feature_map, cmap='gray')
46            plt.axis('off')
47            plt.title('hidden layer')
48            plt.show()

```

第 3 行：用绘图软件做一个 28×28 的图像，在上面写一个数字，我们直接写一个印刷体的 5。

第 4 行：以灰度图像方式读出图像内容。

第 5 行：首先将其形状从二维 28×28 变为一维 784，然后根据每个像素的值进行处理：

值小于 0.5 时取 1，否则取 0。

第 6 行：将其变为 `numpy` 数组，作为一个样本。

第 7、8 行：调用 `load_datasets` 方法，读入训练样本集、验证样本集、测试样本集的内容。

第 9 行：取测试样本集中第 103 个样本作为测试样本。由于我们的模型在训练过程中没有见过测试样本集中的样本，因此可以模拟实际应用中遇到的情况。

第 10 行：将其变为 `[1, 784]` 矩阵形式，可以称之为运行样本集，其中只有一个样本。

第 11 行：定义 `digit` 为最终识别出的 0~9 的数字，取 -1 表示还没有识别结果。

第 12 行：打开 `TensorFlow` 的计算图。

第 13、14 行：以运行模式创建模型（此时无须读入预训练去噪自动编码器的参数和超参数）。

第 15 行：初始化 `TensorFlow` 的 `saver` 对象。

第 16 行：启动 `TensorFlow` 会话来运行程序。

第 17 行：初始化变量。

第 18 行：恢复之前保存的模型参数文件，并初始化模型参数。

第 19 行：求以样本 `X_run` 为输入，在输出层经过 `Softmax` 函数后的计算值，表示每个类别的概率。注意：这里设置 `Dropout` 技术中隐藏层神经元的保留率为 1.0，即所有隐藏层神经元均参与运算，相当于多个随机网络共同投票得出最终结果。

第 20 行：打印出所有类别的概率值。

第 21 行：定义 `max_prob` 记录所有类别最大的概率值。

第 22 行：循环识别结果 `rst` 每个类别的概率。

第 23~25 行：如果最大概率小于当前类别的概率，将当前概率赋给最大概率，识别出的数字等于类别索引值，即对应的 0~9 中的数字。当循环完所有类别后，就能找到概率最大的类别及其对应的数字了。

第 27 行：取出输入层到隐藏层的连接权值矩阵 `W_1`（实际为其转置）。

第 28 行：将第一行形状转为 `28×28` 的图片数据格式。

第 29、30 行：求出隐藏层神经元输出值。注意：这里设置 `Dropout` 技术中隐藏层神经元的保留率为 1.0，即所有隐藏层神经元均参与运算，相当于多个随机网络共同投票得出最终结果。

第 31 行：取出隐藏层输出的原始数据。

第 32 行：由于隐藏层输出数据为 1024 维，我们想显示成一个正方形数据，所以只取前面的 784（`28×28`）维数据。

第 33 行：将隐藏层输出前 784 维变为 `28×28` 的特征图。

第 34 行：将输入样本变为 `28×28` 的黑白图片。

第 35、36 行：初始化 `matplotlib` 绘图库。

第 37~39 行：绘制识别的输入图像，将识别结果显示在标题上。

第 40~43 行：显示输入层到隐藏层第一行连接权值的图像。

第 44~47 行：显示隐藏层输出值图像，因为多层感知器是将原始输入信号变为适合分

类识别的形式，可以视为一种特征学习形式。

第 48 行：同时显示三幅图像。

运行上面的程序，输出结果如图 9.4 和图 9.5 所示。

```
rst:[[ 1.71242209e-04 1.20101404e-05 3.54368922e-06 1.78333512e-03
1.33831973e-05 9.96595800e-01 3.63573622e-06 2.10352555e-05
6.22918946e-04 7.73094478e-04]]
```

图 9.4 堆叠去噪自动编码器运行结果

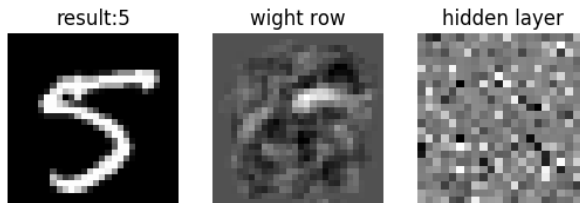


图 9.5 堆叠去噪自动编码器识别结果

从上面的运行结果可以看出，我们的堆叠去噪自动编码器可以正确识别测试样本集上的样本，达到了我们的目的。

9.3 Theano 实现

完成了上面的准备工作之后，我们就可以开始将堆叠自动编码器用于 MNIST 手写数字识别数据集上的手写数字识别任务了。

首先是 MNIST 数据集载入类 `MnistLoader`，代码如下：

```
1 from __future__ import print_function
2 __docformat__ = 'restructuredtext en'
3 import six.moves.cPickle as pickle
4 import gzip
5 import os
6 import sys
7 import timeit
8 import numpy
9 import theano
10 import theano.tensor as T
11
12 class MnistLoader(object):
13     def load_data(self, dataset):
14         data_dir, data_file = os.path.split(dataset)
15         if data_dir == "" and not os.path.isfile(dataset):
16             new_path = os.path.join(
17                 os.path.split(__file__)[0],
18                 "..",
19                 "data",
20                 dataset
21             )
22             if os.path.isfile(new_path) or data_file == 'mnist.pkl.gz':
23                 dataset = new_path
24
25         if (not os.path.isfile(dataset)) and data_file == 'mnist.pkl.gz':
26             from six.moves import urllib
27             origin = (
```

```

28         'http://www.iro.umontreal.ca/~lisa/deep/data/'
29         'mnist/mnist.pkl.gz'
30     )
31     print('Downloading data from %s' % origin)
32     urllib.request.urlretrieve(origin, dataset)
33
34     print('... loading data')
35     # Load the dataset
36     with gzip.open(dataset, 'rb') as f:
37         try:
38             train_set, valid_set, test_set = pickle.load(f,
39                                                         encoding='latin1')
40         except:
41             train_set, valid_set, test_set = pickle.load(f)
42     def shared_dataset(data_xy, borrow=True):
43         data_x, data_y = data_xy
44         shared_x = theano.shared(numpy.asarray(data_x,
45                                                 dtype=theano.config.floatX),
46                                  borrow=borrow)
47         shared_y = theano.shared(numpy.asarray(data_y,
48                                                 dtype=theano.config.floatX),
49                                  borrow=borrow)
50         return shared_x, T.cast(shared_y, 'int32')
51
52     test_set_x, test_set_y = shared_dataset(test_set)
53     valid_set_x, valid_set_y = shared_dataset(valid_set)
54     train_set_x, train_set_y = shared_dataset(train_set)
55
56     rval = [(train_set_x, train_set_y), (valid_set_x, valid_set_y),
57            (test_set_x, test_set_y)]
58     return rval

```

第 13 行：定义数据载入接口，参数为 MNIST 手写数字识别数据集文件。

第 14~33 行：MNIST 手写数字识别数据集文件如果存在则打开该文件，如果不存在则从指定网址下载该文件。

第 52~54 行：将训练样本集、验证样本集、测试样本集定义为 Theano 的共享变量，以便在 Theano 预编译函数中使用。

第 56~58 行：以指定格式返回 MNIST 手写数字识别数据内容。

堆叠去噪自动编码器训练入口类的代码如下：

```

1 from sda_engine import SdAEngine
2
3 if __name__ == '__main__':
4     engine = SdAEngine()
5     engine.train()

```

运行堆叠去噪自动编码器训练程序，程序会打印出如下内容：

```

create SdAEngine
... loading data
... building the model
Yantao: *****
... getting the pretraining functions
... pre-training the model
Pre-training layer 0, epoch 0, cost 71.648650
Pre-training layer 0, epoch 1, cost 63.687814
Pre-training layer 0, epoch 2, cost 62.291742
Pre-training layer 0, epoch 3, cost 61.579868
Pre-training layer 0, epoch 4, cost 61.153720
Pre-training layer 0, epoch 5, cost 60.873762
Pre-training layer 0, epoch 6, cost 60.607693
Pre-training layer 0, epoch 7, cost 60.413270
Pre-training layer 0, epoch 8, cost 60.205539
Pre-training layer 0, epoch 9, cost 60.040651
Pre-training layer 0, epoch 10, cost 59.906401
Pre-training layer 0, epoch 11, cost 59.726363
Pre-training layer 0, epoch 12, cost 59.564314
Pre-training layer 0, epoch 13, cost 59.472843

```

以上是预训练去噪自动编码器过程的输出内容。在我的虚拟机上，上面的过程会运行 10 个小时左右，进入调优状态后的输出内容如下：

```
The pretraining for file mnist.pkl.gz run for 583m
... getting finetuning functions
... finetuning the model
.....
epoch 5001, minibatch 50/600, test error of best model 15.52%
.....
Optimization complete with best validation score of 0.96% on iteration 687583, w
ith test performance 0.98%
The training code for file mnist.pkl.gz run for 695m
```

为节省篇幅，只截取了部分输出内容，以及最终的打印结果。

大家可以看到，在训练速度和识别精度方面，堆叠去噪自动编码器与之前介绍的卷积神经网络会有些差距，这就说明不同的网络适用于不同的任务。在图像识别领域，首选卷积神经网络；而在图像搜索等领域，堆叠去噪自动编码器的应用效果更佳。

第 10 章

受限玻尔兹曼机

在本章中，我们将讨论受限玻尔兹曼机网络，这种网络采用能量函数进行定义，对比散度算法（CD-K）进行训练，广泛应用于深度学习的特征工程中。而且受限玻尔兹曼机还可以通过堆叠形成深度信念网络，这种网络是深度学习算法兴起的起点，具有广泛的应用价值，历史意义也非常大。

在本章中，我们会先介绍受限玻尔兹曼机的数学原理，然后会以 Theano 框架为基础，实现一个简单的受限玻尔兹曼机，并将其用于图像特征获取工作，使我们对受限玻尔兹曼机有一个感性认识。

10.1 受限玻尔兹曼机原理

10.1.1 网络架构

我们目前讨论的神经网络，虽然学习算法不同，但架构基本还是相同的，都是分层网络，即神经元按层进行组织，层内神经元无连接，层间神经元有连接。在本章中，将讨论一种非常不同的神经网络，这种神经网络由没有层次关系的神经元全连接网络进化而来。

这种网络起源于 Holpfield 网络，我们可以给它定义一个能量函数，神经网络的学习任务就是使能量函数达到最小值。这种网络典型的应用是担货郎问题，即有 N 个地点，每个地点间都有道路相通，担货郎必须把货物送到每个地点。通过 Holpfield 网络，可以有效地找到最佳路径。但是即使是二值（神经元只能处在 0 或 1 状态），全连接网络也有 2^N 个状态，要从这些状态中找到能量函数的最小值，难度相当大。

与此同时，根植于统计力学模型的玻尔兹曼机也开始流行起来。在这种网络中，神经

元的输出只有激活和未激活两种状态，用 0 或 1 来表示，各个神经元的输出值由概率统计模型给出。典型的玻尔兹曼机的网络模型如图 10.1 所示。

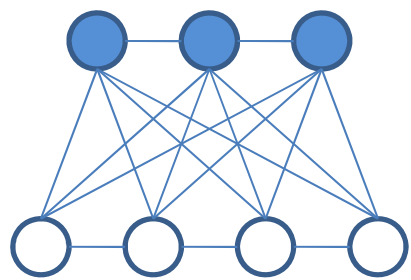


图 10.1 全连接网络

从实践来看，玻尔兹曼机具有强大的非监督学习能力，可以发现数据中的潜在规则，从理论上讲，它非常适合应用于数据挖掘领域。但是由于是全连接网络，导致这种网络的训练时间非常长，没有高效的学习算法，直接制约了这种网络的应用。

后来 Smolensky 引入了受限玻尔兹曼机模型，其主要思想就是去掉玻尔兹曼机中的层内连接。受限玻尔兹曼机具有一个可见层和一个隐藏层，层内神经元间无连接，层间神经元全连接，如图 10.2 所示。

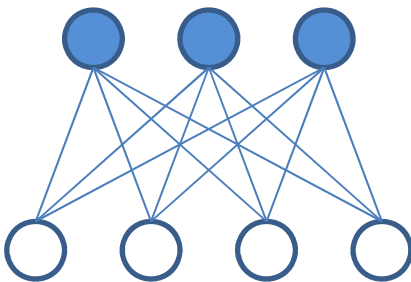


图 10.2 受限玻尔兹曼机示意图

在受限玻尔兹曼机中，输入信号通过可见层输入网络中，传播到隐藏层后，各隐藏层神经元的激活是互相独立的。同理，在给定隐藏层信号后，反向传播到可见层时，可见层神经元的激活也具有独立性。这从理论上证明，只要这种网络结构的隐藏层神经元节点足够多，受限玻尔兹曼机可以拟合任意离散分布。虽然在理论上受限玻尔兹曼机很好，但是由于一直没有高效的学习算法，所以它并没有得到广泛应用。但是深度学习之父 Hinton 在 2002 年提出了对比散度（CD）算法，使受限玻尔兹曼机具备了快速学习的能力。从此，受限玻尔兹曼机得到了广泛应用，出现了各种对比散度算法的变种，使算法收敛性更高。与此同时，玻尔兹曼机在分类、回归、降噪、高维时间序列分析、图像特征提取、协同过滤等方面得到了广泛应用。而且 Science 子刊上还发表了利用受限玻尔兹曼机分析非结构化病例信息，从而进行医学诊断的例子，并成功应用于癌症早期筛查，这表明受限玻尔兹曼机在非结构化数据处理方面也有实用价值。另外，Hinton 在 2006 年提出将受限玻尔兹曼机堆叠起来，形成深度信念网络，通过逐层训练受限玻尔兹曼机网络，将训练好的受限玻尔

兹曼机网络堆叠成深度学习网络，可以得到非常好的初始参数值，有效地解决了大型神经网络训练速度慢的问题，是当前的研究热点之一。

在介绍了这么多受限玻尔兹曼机的基本情况之后，我们来具体介绍一下它的数学原理。

10.1.2 能量模型

受限玻尔兹曼机的架构可用图 10.3 表示。

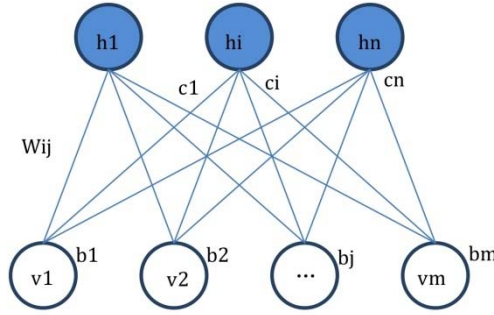


图 10.3 受限玻尔兹曼机架构图

图中 v 代表可见层，共有 m 个节点，下标用 j 表示，其神经元对应的偏移量为 b_j 。 h 代表隐藏层，共有 n 个节点，下标用 i 表示，偏移量用 c_i 表示。用 W_{ij} 表示可见层和隐藏层间的连接权值。同时可见层神经元： $\forall j \ v_j \in \{0,1\}$ ，对于隐藏层： $\forall i \ h_i \in \{0,1\}$ 。

定义受限玻尔兹曼机的能量函数为：

$$E(\mathbf{v}, \mathbf{h}|\boldsymbol{\theta}) = -\sum_{i=1}^n \sum_{j=1}^m v_j W_{ij} h_i - \sum_{j=1}^m b_j v_j - \sum_{i=1}^n c_i h_i \quad (1)$$

式中， $\boldsymbol{\theta}$ 为受限玻尔兹曼机的参数，即 $\boldsymbol{\theta} = \{W_{ij}, b_j, c_i\}$ 。

可见层和隐藏层的联合概率分布为：

$$P(\mathbf{v}, \mathbf{h}|\boldsymbol{\theta}) = \frac{e^{-E(\mathbf{v}, \mathbf{h}|\boldsymbol{\theta})}}{Z} \quad (2)$$

式中， Z 为归一化因子，其定义为：

$$Z(\boldsymbol{\theta}) = \sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h}|\boldsymbol{\theta})} \quad (3)$$

熟悉机器学习算法的人都知道，我们需要知道的实际上是输入样本的概率分布，也就是式（2）中 P 的边际分布，也称为似然函数，定义如下：

$$P(\mathbf{v}|\boldsymbol{\theta}) = \frac{1}{Z(\boldsymbol{\theta})} \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h}|\boldsymbol{\theta})} \quad (4)$$

因为需要计算归一化因子 $Z(\theta)$ ，而这需要 2^{m+n} 次运算，对于高维问题，即使我们可以通过算法得出网络的参数，但是运算量过大，因此这个公式在实际应用中也不能直接应用。

但是，由于受限玻尔兹曼机具有层间全连接、层内无连接的特点，当我们将输入信号输入到可见层时，可见层将决定隐藏层各神经元的状态，而且由于层内无连接，此时隐藏层神经元的激活状态是条件独立的，隐藏层第 j 个神经元为激活状态的概率为：

$$P(h_i = 1|\mathbf{v}, \theta) = \sigma\left(\sum_{j=1}^m W_{ij}v_j + c_i\right) \quad (5)$$

如果隐藏层状态完全确定时，可见层第 i 个神经元的激活状态也是条件独立的，其公式为：

$$P(v_j = 1|\mathbf{h}, \theta) = \sigma\left(\sum_{i=1}^n W_{ij}h_i + b_j\right) \quad (6)$$

下面来定义受限玻尔兹曼机的对数似然函数：

$$\begin{aligned} \log \mathcal{L}(\theta|\mathbf{v}) &= \log P(\mathbf{v}|\theta) \\ &= \log \frac{1}{Z} \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \quad \textcircled{1} \\ &= \log \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} - \log \sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \quad \textcircled{2} \end{aligned} \quad (7)$$

在①处，为求出 \mathbf{v} 在某个状态下的概率，我们先用 \mathbf{v} 在某个状态下对所有 \mathbf{h} 状态的能量函数求和，然后再除以总的能量函数之和，即归一化因子 Z ，从而得到 $P(\mathbf{v}|\theta)$ 。在②处，我们将归一化因子 Z 的定义代入，同时将对数内相除变为对数相减，就可以得到结果。

下面来求对数似然函数的导数：

$$\frac{\partial \log \mathcal{L}(\theta|\mathbf{v})}{\partial \theta} = \frac{\partial}{\partial \theta} \left(\log \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \right) - \frac{\partial}{\partial \theta} \left(\log \sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \right) \quad (8)$$

由于此式比较复杂，需要分别对第一项和第二项进行求导，考虑到大家对数学公式的熟悉程度，我们将一步一步进行推导，中间用到的数学公式我们会在文中列出。

首先对式中的第一项进行求导：

$$\frac{\partial}{\partial \theta} \left(\log \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \right) = \frac{\partial (\log \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})})}{\partial \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}} \frac{\partial \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}}{\partial E(\mathbf{v}, \mathbf{h})} \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta} \quad \textcircled{1}$$

$$= \frac{1}{\sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}} \left(- \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \right) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \theta} \quad \textcircled{2}$$

$$= - \left(\sum_h \frac{e^{-E(v,h)}}{\sum_h e^{-E(v,h)}} \right) \frac{\partial E(v,h)}{\partial \theta} \quad (9)$$

在①处使用了高等数学中的链式求导法则，对复合函数求导可以使用以下定理：如果 $u=g(x)$ 在点 x 可导，而 $y=f(u)$ 在点 $u=g(x)$ 可导，则复合函数 $y=f[g(x)]$ 在点 x 处可导，且其导数为：

$$\frac{dy}{dx} = \frac{dy}{du} \frac{du}{dx} \quad (10)$$

在②中，我们用到了常用初等函数的导数公式：

$$(\log x)' = \frac{1}{x} \quad (11)$$

$$(e^x)' = e^x \quad (12)$$

还用到了函数和的导数等于函数导数的和。

在③处，需要将②处的第一项移到第二项中，不能约去，所以得到式（9）的最终结果。我们在概率论与数理统计中学过贝叶斯定理，如下：

$$P(y|x) = \frac{P(y,x)}{P(x)} \quad (13)$$

贝叶斯定理表明，在 x 发生条件下 y 的概率，可以表示为 x 和 y 的联合概率除以 x 的概率。

所以在给定可见层条件下隐藏层的条件概率分布，用贝叶斯定理可以表示为：

$$P(h|v) = \frac{P(v,h)}{P(v)} = \frac{\frac{1}{Z} e^{-E(v,h)}}{\frac{1}{Z} \sum_h e^{-E(v,h)}} = \frac{e^{-E(v,h)}}{\sum_h e^{-E(v,h)}} \quad (14)$$

在推导中用到了式（2）中 $P(v,h)$ 的定义。

将式（14）代入式（9），得到：

$$\frac{\partial}{\partial \theta} \left(\log \sum_h e^{-E(v,h)} \right) = - \left(\sum_h \frac{e^{-E(v,h)}}{\sum_h e^{-E(v,h)}} \right) \frac{\partial E(v,h)}{\partial \theta} = - \left(\sum_h P(h|v) \right) \frac{\partial E(v,h)}{\partial \theta} \quad (15)$$

到目前为止，我们已经成功求出了式（8）第一项的内容，该公式变为：

$$\frac{\partial \log \mathcal{L}(\theta|v)}{\partial \theta} = - \left(\sum_h P(h|v) \right) \frac{\partial E(v,h)}{\partial \theta} - \frac{\partial}{\partial \theta} \left(\log \sum_{v,h} e^{-E(v,h)} \right) \quad (16)$$

下面我们来求式（16）的第二项。

$$\begin{aligned}
\frac{\partial}{\partial \boldsymbol{\theta}} \left(\log \sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \right) &= \frac{\partial (\log \sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})})}{\partial \sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}} \frac{\partial \sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}}{\partial e^{-E(\mathbf{v}, \mathbf{h})}} \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \boldsymbol{\theta}} \\
&= \frac{1}{\sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}} \left(- \sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \right) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \boldsymbol{\theta}} \\
&= - \left(\sum_{\mathbf{v}, \mathbf{h}} \frac{e^{-E(\mathbf{v}, \mathbf{h})}}{\sum_{\mathbf{v}, \mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}} \right) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \boldsymbol{\theta}} \\
&= - \left(\sum_{\mathbf{v}, \mathbf{h}} \frac{e^{-E(\mathbf{v}, \mathbf{h})}}{Z} \right) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \boldsymbol{\theta}} \\
&= - \left(\sum_{\mathbf{v}, \mathbf{h}} P(\mathbf{v}, \mathbf{h}) \right) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \boldsymbol{\theta}} \tag{17}
\end{aligned}$$

因此对数似然函数对参数求导公式的最终形式为：

$$\frac{\partial \log \mathcal{L}(\boldsymbol{\theta} | \mathbf{v})}{\partial \boldsymbol{\theta}} = - \left(\sum_{\mathbf{h}} P(\mathbf{h} | \mathbf{v}) \right) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \boldsymbol{\theta}} + \left(\sum_{\mathbf{v}, \mathbf{h}} P(\mathbf{v}, \mathbf{h}) \right) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial \boldsymbol{\theta}} \tag{18}$$

根据对受限玻尔兹曼机的定义，参数 $\boldsymbol{\theta} = \{W_{ij}, b_j, c_i\}$ ，下面我们分别对这些实际参数进行求导。

先对 W_{ij} 求导：

$$\frac{\partial \log \mathcal{L}(\boldsymbol{\theta} | \mathbf{v})}{\partial W_{ij}} = - \left(\sum_{\mathbf{h}} P(\mathbf{h} | \mathbf{v}) \right) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial W_{ij}} + \left(\sum_{\mathbf{v}, \mathbf{h}} P(\mathbf{v}, \mathbf{h}) \right) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial W_{ij}} \tag{19}$$

连接权值 W_{ij} 的导数为：

$$\frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial W_{ij}} = v_j h_i \tag{20}$$

我们先来求式 (18) 的第一项：

$$\begin{aligned}
\left(\sum_{\mathbf{h}} P(\mathbf{h} | \mathbf{v}) \right) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial W_{ij}} &= \left(\sum_{\mathbf{h}} P(\mathbf{h} | \mathbf{v}) \right) v_j h_i \\
&= \left(\sum_{\mathbf{h}} \prod_{k=1}^n p(h_k | \mathbf{v}) \right) v_j h_i \\
&= P(H_i = 1 | \mathbf{v}) v_j
\end{aligned} \tag{21}$$

下面我们来求式（18）的第二项：

$$\begin{aligned}\left(\sum_{\mathbf{v}, \mathbf{h}} p(\mathbf{v}, \mathbf{h})\right) \frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial W_{ij}} &= \left(\sum_{\mathbf{v}} p(\mathbf{v}) \sum_{\mathbf{h}} p(\mathbf{h}|\mathbf{v})\right) v_j h_i \\ &= \sum_{\mathbf{v}} p(\mathbf{v}) P(H_i = 1|\mathbf{v}) v_j\end{aligned}\quad (22)$$

则式（19）的最终结果为：

$$\frac{\partial \log \mathcal{L}(\boldsymbol{\theta}|\mathbf{v})}{\partial W_{ij}} = -P(H_i = 1|\mathbf{v}) v_j + \sum_{\mathbf{v}} p(\mathbf{v}) P(H_i = 1|\mathbf{v}) v_j \quad (23)$$

如果采取在线学习模式，即每个训练样本均更新神经网络的参数，式 21 即可作为权值调整值来使用，但是由于样本的随机性，这样的参数调整效率会比较低，因此实际中使用更多的是迷你批次学习模式。假设设置批次学习样本为：

$$S = \{v_1, v_2, \dots, v_l\}$$

求导就变成对迷你批次内所有样本求导，先求出这些导数之和，再除以迷你批次中的样本数，如下：

$$\begin{aligned}\frac{1}{\ell} \sum_{\mathbf{v} \in S} \frac{\partial \log \mathcal{L}(\boldsymbol{\theta}|\mathbf{v})}{\partial W_{ij}} &= \frac{1}{\ell} \sum_{\mathbf{v} \in S} \left[-E_{p(\mathbf{h}|\mathbf{v})} \left[\frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial W_{ij}} \right] + E_{p(\mathbf{h}, \mathbf{v})} \left[\frac{\partial E(\mathbf{v}, \mathbf{h})}{\partial W_{ij}} \right] \right] \\ &= \frac{1}{\ell} \sum_{\mathbf{v} \in S} \left[E_{p(\mathbf{h}|\mathbf{v})} [v_j h_i] - E_{p(\mathbf{h}, \mathbf{v})} [v_j h_i] \right] \\ &= \langle v_j h_i \rangle_{p(\mathbf{h}|\mathbf{v})q(\mathbf{v})} - \langle v_j h_i \rangle_{p(\mathbf{h}, \mathbf{v})}\end{aligned}\quad (24)$$

这就是在迷你批次学习下对连接权值的导数公式。式中， q 代表经验分布。

通常对数似然函数对连接权值的导数，有如下关系：

$$\frac{\partial \log \mathcal{L}(\boldsymbol{\theta}|\mathbf{v})}{\partial W_{ij}} \propto \langle v_j h_i \rangle_{\text{data}} - \langle v_j h_i \rangle_{\text{model}}$$

即对数似然函数对连接权值的导数正比于实际数据分布与模型数据分布之差。

同理，我们可以求出对数似然函数对可见层偏移量 b_j 的导数，如下：

$$\frac{\partial \log \mathcal{L}(\boldsymbol{\theta}|\mathbf{v})}{\partial b_j} = v_j - \sum_{\mathbf{v}} p(\mathbf{v}) v_j \quad (25)$$

对数似然函数对隐藏层偏移量 c_i 的导数，如下：

$$\frac{\partial \log \mathcal{L}(\boldsymbol{\theta}|\mathbf{v})}{\partial c_i} = P(H_i = 1|\mathbf{v}) - \sum_{\mathbf{v}} p(\mathbf{v}) P(H_i = 1|\mathbf{v}) \quad (26)$$

式 (24)、式 (25)、式 (26) 中的第二项, 可以通过 Gibbs 采样得到, 但是需要运行受限玻尔兹曼机很长时间, 即从可见层到隐藏层, 再从隐藏层到可见层, 反复运行, 直到达到静止点, 这就是马可夫链蒙特卡罗方法。但是使用这种方法的运算量非常大, 受限于此, 在 2006 年以前, 受限玻尔兹曼机很少应用在实际问题中。在 2006 年, 深度学习之父 Hinton 提出了 CD-K 算法, 很好地解决了这一问题, 才使受限玻尔兹曼机具有了应用价值, 并且直接促使深度信念网络的广泛使用。

10.1.3 CD-K 算法

CD-K 算法认为, 当我们使用训练数据初始化时, 将 v_0 加载到可见层, 仅需要使用 k (通常 $k=1$) 步吉布斯采样, 就可以得到足够好的近似值。在 CD-K 算法中, 先将可见层神经元设置成一个训练样本, 利用下式计算隐藏层神经元的状态:

$$P(h_i = 1 | \mathbf{v}, \boldsymbol{\theta}) = \sigma \left(\sum_{j=1}^m v_j W_{ij} + c_i \right) \quad (27)$$

当所有隐藏层神经元的状态确定之后, 根据隐藏层神经元状态重构出可见层神经元状态, 利用下式进行计算:

$$P(v_j = 1 | \mathbf{h}, \boldsymbol{\theta}) = \sigma \left(\sum_{i=1}^n W_{ij} h_i + b_j \right) \quad (28)$$

使用梯度上升算法进行学习, 则各个参数的更新准则为:

$$\Delta W_{ij} = \alpha (\langle v_j h_i \rangle_{\text{data}} - \langle v_j h_i \rangle_{\text{reconstruct}}) \quad (29)$$

式中, α 为学习率。

同理, 可以得到其他参数的更新规则:

$$\Delta b_j = \alpha (\langle v_j \rangle_{\text{data}} - \langle v_j \rangle_{\text{reconstruct}}) \quad (30)$$

$$\Delta c_i = \alpha (\langle h_i \rangle_{\text{data}} - \langle h_i \rangle_{\text{reconstruct}}) \quad (31)$$

在构建受限玻尔兹曼机时, 通常设置可见层神经元数为所研究问题的维度, 只需要设置隐藏层神经元数量即可。

设置可见层为:

$$\forall j \quad v_j = \{0, 1\}$$

$$\mathbf{b} \in \mathbb{R}^m \quad \text{可见层偏移量}$$

$$\forall i \quad h_i = \{0, 1\}$$

$\mathbf{c} \in \mathbb{R}^n$ 隐藏层偏移量

连接权值矩阵： $\mathbf{W} \in \mathbb{R}^{m \times n}$

基本 CD-K 算法如下。

(1) 输入：训练样本 $\mathbf{x}^{(i)}$ ，隐藏层神经元数 n ，学习率 α ，最大训练周期 T 。

(2) 输出：连接权值矩阵 \mathbf{W} ，可见层偏置向量 \mathbf{b} ，隐藏层偏置向量 \mathbf{c} 。

(3) 训练算法。

初始化：令可见层神经元的初始状态 $\mathbf{v}^{(1)} = \mathbf{x}^{(i)}$ ， \mathbf{W} 、 \mathbf{b} 、 \mathbf{c} 取随机较小的数值。

for t=1 to T

for i=1, 2, ..., n（对所有隐藏层神经元）

计算： $P(h_i^{(1)} | \mathbf{v}^{(1)}) = \sigma(\sum_{j=1}^m v_j^{(1)} W_{ij})$

采样：如果 $P(h_i^{(1)} | \mathbf{v}^{(1)}) > 0.5$ ，则 $h_i^{(1)} = 1$ ，否则 $h_i^{(1)} = 0$

for j=1, 2, ..., m（对所有可见层神经元）

计算： $P(v_j^{(2)} = 1 | \mathbf{h}^{(1)}) = \sigma(\sum_{i=1}^n W_{ij} h_i^{(1)})$

采样：如果 $P(v_j^{(2)} = 1 | \mathbf{h}^{(1)}) > 0.5$ ，则 $v_j^{(2)} = 1$ ，否则 $v_j^{(2)} = 0$

for i=1, 2, ..., n（对所有隐藏层神经元）

计算： $P(h_i^{(2)} | \mathbf{v}^{(2)}) = \sigma(\sum_{j=1}^m v_j^{(2)} W_{ij})$

更新各个参数值：

$$\mathbf{W} = \mathbf{W} + \alpha (P(\mathbf{h}^{(1)} | \mathbf{v}^{(1)}) (\mathbf{v}^{(1)})^T - P(\mathbf{h}^{(2)} = 1 | \mathbf{v}^{(2)}) (\mathbf{v}^{(2)})^T)$$

$$\mathbf{b} = \mathbf{b} + \alpha (\mathbf{v}^{(1)} - \mathbf{v}^{(2)})$$

$$\mathbf{c} = \mathbf{c} + \alpha (P(\mathbf{h}^{(1)} = 1 | \mathbf{v}^{(1)}) - P(\mathbf{h}^{(2)} = 1 | \mathbf{v}^{(2)}))$$

以上就是最基本的 CD-K 算法，从算法描述中可以看出，CD-K 算法是针对二值网络的快速算法，但是如果将循环中的采样步骤省略，就可以处理连续信息。

在实际应用中，研究人员通常采用持续对比散度（Persistent Contrastive Divergence, PCD）算法，其与普通 CD-K 算法的主要区别如下：

- ❑ 不使用训练数据初始化吉布斯采样的马可夫链。
- ❑ 学习率小且不断衰减。

10.2 受限玻尔兹曼机 TensorFlow 实现

首先来看受限玻尔兹曼机引擎类的构造函数，代码如下：

```
1 def __init__(self, name='rbm'):
2     self.datasets_dir = 'datasets/'
3     self.random_seed = 1 # 用于测试目的，使每次生成的随机数相同
4     self.name = name
5     #self.loss_func = loss_func
6     self.learning_rate = 0.0001
7     self.num_epochs = 50
8     self.batch_size = 128
9     self.regtype = 'l2'
10    self.regcoef = 0.00001
11    self.num_hidden = 250
12    self.visible_unit_type = 'bin'
13    self.gibbs_sampling_steps = 3
14    self.stddev = 0.1
15    self.W = None
16    self.bh_ = None
17    self.bv_ = None
18    self.w_upd8 = None
19    self.bh_upd8 = None
20    self.bv_upd8 = None
21    self.cost = None
22    self.input_data = None
23    self.hrand = None
24    self.vrand = None
25    self.tf_graph = tf.Graph()
26    self.n = 784 # 28×28 黑白图片
```

第 2 行：定义数据集文件存放路径。

第 3 行：定义随机数生成种子，以种子方式生成的随机数每次生成的一样，这样便于对神经网络的调试。

第 4 行：定义网络的名称，因为在深度信念网络中，我们会将预训练好的受限玻尔兹曼机堆叠起来，为了区分不同的受限玻尔兹曼机，需要给其起不同的名字。

第 6 行：定义学习率。

第 7 行：定义完整学习整个训练样本集的遍数。

第 8 行：迷你批次中的样本数量。

第 9 行：调整项类型，我们在这里使用 L2 调整项，即权值衰减项。

第 10 行：定义 L2 调整项（权值衰减）的系数。

第 11 行：定义隐藏层神经元数量。

第 12 行：定义可见层神经元类型，经典受限玻尔兹曼机的可见层是二进制类型的，如本例所示。而现在经过扩展的受限玻尔兹曼机可以支持连续型变量。

第 13 行：吉布斯采样次数。

第 14 行：定义正态分布中的标准差。

- 第 15 行：定义可见层和隐藏层之间的连接权值。
- 第 16 行：定义隐藏层的偏置值矩阵。
- 第 17 行：定义可见层的偏置值矩阵。
- 第 18 行：权值更新时 TensorFlow 计算图中的节点。
- 第 19 行：隐藏层偏置值更新时对应 TensorFlow 计算图中的节点。
- 第 20 行：可见层偏置值更新时对应 TensorFlow 计算图中的节点。
- 第 21 行：代价函数在 TensorFlow 计算图中的节点。
- 第 22 行：输入信号变量。
- 第 23 行：隐藏层对应的 placeholder。
- 第 24 行：可见层对应的 placeholder。
- 第 25 行：本模型对应的 TensorFlow 计算图。
- 第 26 行：输入向量维度。

下面来看 MNIST 数据集载入，代码如下：

```
1 def load_datasets(self):
2     mnist = input_data.read_data_sets(self.datasets_dir,
3         one_hot=True)
4     X_train = mnist.train.images
5     y_train = mnist.train.labels
6     X_validation = mnist.validation.images
7     y_validation = mnist.validation.labels
8     X_test = mnist.test.images
9     y_test = mnist.test.labels
10    return X_train, y_train, X_validation, y_validation, \
11        X_test, y_test, mnist
```

第 2、3 行：调用 TensorFlow 的 `input_data` 的 `read_data_sets` 方法，第一个参数为数据集存放路径，第二个参数是标签集的格式。在原始 MNIST 数据集中，我们知道每个样本是 28×28 的黑白图片，对应的是 0~9 的数字标签，所以其格式为：[...784 (28×28) 像素点的值...] [3]。其中，第一项为 784 (28×28) 个 0~1 的浮点数，0 代表黑色，1 代表白色；第二项的“3”代表这个样本是数字 3。为了后续处理方便，我们将标签 [3] 改为 one-hot 形式，因为标签代表 0~9 的数字，所以标签集为 10 维向量，每维上取值为 0 代表不是这个对应位置的数字，取值为 1 代表是这个对应位置的数字。其中，只有一维可以取 1，因此称之为 one-hot，还以上面的例子为例，标签集的格式就变为：[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]，因为第四位为 1，所以代表这个样本是数字 3。

第 4 行：取出训练样本集输入信号集 `X_train`，其为设计矩阵形式，每一行代表一个样本，行数为训练样本集中的样本数量。在这个例子中，就 $X_{train} \in \mathbb{R}^{55000 \times 784}$ ，其中训练样本集中有 55000 个样本，每个样本是 784 (28×28) 维的图片。

第 5 行：取出训练样本集标签集 `y_train`，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为训练样本集中的样本数量。在这个例子中，就 $y_{train} \in \mathbb{R}^{55000 \times 10}$ ，其中训练样本集中有 55000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 6 行：取出验证样本集输入信号集 $X_{\text{validation}}$ ，其为设计矩阵形式，每一行代表一个样本，行数为验证样本集中的样本数量。在这个例子中，就 $X_{\text{validation}} \in \mathbb{R}^{5000 \times 784}$ ，其中验证样本集中有 5000 个样本，每个样本是 784（28×28）维的图片。根据前面我们的讨论可以知道，在训练过程中，为了防止模型出现过拟合，模型的泛化能力降低（模型在训练样本集达到非常高的精度，但是在未见过的测试样本集或实际应用中，精度反而不高），通常会采用 **Early Stopping** 策略，就是在逻辑回归模型训练过程中，只用训练样本集对模型进行训练，每隔一定的时间间隔，计算模型在未见过的验证样本集上识别的精度，并记录迄今为止在验证样本集上取得的最高精度。我们会发现，在训练初期，验证样本集上的识别精度会稳步提高，但是到了一定阶段之后，验证样本集上的识别精度就不会再明显提高了，甚至开始逐渐下降，这就说明模型出现了过拟合，这时就可以停止模型训练，将在验证样本集上取得最佳识别精度的模型参数作为模型最终的参数。综上所述，验证样本集主要用于防止模型出现过拟合，为 **Early Stopping** 算法提供终止依据。

第 7 行：取出验证样本集标签集 $y_{\text{validation}}$ ，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为训练样本集中的样本数量。在这个例子中，就 $y_{\text{validation}} \in \mathbb{R}^{5000 \times 10}$ ，其中验证样本集中有 5000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 8 行：取出测试样本集输入信号集 X_{test} ，其为设计矩阵形式，每一行代表一个样本，行数为训练样本集中的样本数量。在这个例子中，就 $X_{\text{test}} \in \mathbb{R}^{10000 \times 784}$ ，其中训练样本集中有 10000 个样本，每个样本是 784（28×28）维的图片。测试样本集主要用于模型训练结束后对模型性能进行评估。由于模型没有见过测试样本集中的样本，可以模拟模型在实际部署之后的情况，模型在测试样本集上的识别精度，基本可以视为模型在实际应用中可以达到的精度。

第 9 行：取出测试样本集标签集 y_{test} ，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为测试样本集中的样本数量。在这个例子中，就 $y_{\text{test}} \in \mathbb{R}^{10000 \times 10}$ ，其中测试样本集中有 10000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 10、11 行：返回训练样本集输入信号集、训练样本集标签集、验证样本集输入信号集、验证样本集标签集、测试样本集输入信号集、测试样本集标签集。

下面来看受限玻尔兹曼机模型构造程序，代码如下：

```
1 def build_model(self):
2     print('Build RBM Model')
3     self.X = tf.placeholder(shape=[None, self.n], dtype=tf.float32, name='X')
4     self.hrand = tf.placeholder(shape=[None, self.num_hidden],
5                                 dtype=tf.float32, name='h')
6     self.vrand = tf.placeholder(shape=[None, self.n],
7                                 dtype=tf.float32, name='v')
8     self.y = tf.placeholder(shape=[None, 10], dtype=tf.float32, name='y')
9     self.keep_prob = tf.placeholder(dtype=tf.float32, name='keep_prob')
10    #
11    self.W = tf.Variable(tf.truncated_normal(shape=[self.n, self.num_hidden],
12                                              mean=0.0, stddev=0.1), name='W')
```



```

13 self.bh_ = tf.Variable(tf.constant(0.1, shape=[self.num_hidden]),
14                        name='bh')
15 self.bv_ = tf.Variable(tf.constant(0.1, shape=[self.n], name='bv'))
16 #
17 self.encode, _ = self.sample_hidden_from_visible(self.X)
18 self.reconstruction = self.sample_visible_from_hidden(
19     self.encode, self.n)
20 hprob0, hstate0, vprob, hprob1, hstate1 = self.gibbs_sampling_step(
21     self.X, self.n)
22 self.vprob = vprob
23 positive = self.compute_positive_association(self.X,
24                                              hprob0, hstate0)
25 nn_input = vprob
26 for step in range(self.gibbs_sampling_steps - 1):
27     hprob, hstate, vprob, hprob1, hstate1 = self.gibbs_sampling_step(
28         nn_input, self.n)
29     nn_input = vprob
30 negative = tf.matmul(tf.transpose(vprob), hprob1)
31 #
32 self.w_upd8 = self.W.assign_add(
33     self.learning_rate * (positive - negative) / self.batch_size)
34 self.bh_upd8 = self.bh_.assign_add(tf.multiply(self.learning_rate,
35         tf.reduce_mean(tf.subtract(hprob0, hprob1), 0)))
36 self.bv_upd8 = self.bv_.assign_add(tf.multiply(self.learning_rate,
37         tf.reduce_mean(tf.subtract(self.X, vprob), 0)))
38 clip_inf = tf.clip_by_value(vprob, 1e-10, float('inf'))
39 clip_sup = tf.clip_by_value(1 - vprob, 1e-10, float('inf'))
40 loss = - tf.reduce_mean(tf.add(
41     tf.multiply(self.X, tf.log(clip_inf)),
42     tf.multiply(tf.subtract(1.0, self.X),
43         tf.log(clip_sup))))
44 self.cost = loss + self.regcoef*(tf.nn.l2_loss(self.W) +
45     tf.nn.l2_loss(self.bh_) + tf.nn.l2_loss(self.bv_))

```

第 3 行：定义用于存放输入信号的 placeholder，其为包含一个迷你批次的设计矩阵，第一维是迷你样本集中的序号，第二维为样本特征值向量。

第 4、5 行：定义初始化隐藏层为随机数时用的 placeholder，第一维为迷你批次中的样本序号，第二维为隐藏层神经元数。

第 6、7 行：定义初始化可见层为随机数时用的 placeholder，第一维为迷你批次中的样本序号，第二维为可见层神经元数。

第 8 行：定义正确分类结果标签集的 placeholder，由于受限玻尔兹曼机为非监督学习，所以这个变量目前没有用到。

第 11、12 行：定义可见层与隐藏层之间的连接权值矩阵 W，用均值为 0.0、标准差为 0.1 的正态分布随机数进行初始化。

第 13、14 行：定义隐藏层神经元偏置值 bh_，用常数 0.1 进行初始化。

第 15 行：定义可见层神经元偏置值 bv_，用常数 0.1 进行初始化。

第 17 行：定义受限玻尔兹曼机编码过程，调用本类 sample_hidden_from_visible 方法，通过可见层状态求出隐藏层状态，可以视为对原始信号进行编码，或者提取出其中的特征。

第 18、19 行：定义重建算子，调用本类 sample_visible_from_hidden 方法，由隐藏层状

态求出可见层状态，相当于通过特征恢复出原始信号。

第 20、21 行：定义吉布斯采样算子，调用本类 `gibbs_sampling_step` 方法，即先由可见层求出隐藏层状态 0，再由隐藏层状态求出可见层状态，最后由新的可见层状态求出隐藏层状态 1，返回值 `hprob0` 和 `hstate0` 分别代表第一次求出的隐藏层状态对应的概率值和由概率值转化而来的二进制状态，`vprob` 代表可见层概率值，`hprob1` 和 `hstate1` 分别代表由上一状态求出的隐藏层状态对应的概率值和由概率值转化而来的二进制状态。

第 22 行：将可见层概率值定义为 TensorFlow 算子。

第 23、24 行：计算正向状态，即第一次由可见层到隐藏层时，可见层状态与隐藏层状态的点积。

第 25 行：将当前可见层概率值作为输入参数。

第 26 行：循环第 27~29 行操作，执行 `self.gibbs_sampling_steps - 1` 次吉布斯采样。

第 27、28 行：先由可见层求出隐藏层状态 0，再由隐藏层状态求出可见层状态，最后由新的可见层状态求出隐藏层状态 1，返回值 `hprob` 和 `hstate` 分别代表第一次求出的隐藏层状态对应的概率值和由概率值转化而来的二进制状态，`vprob` 代表可见层概率值，`hprob1` 和 `hstate1` 分别代表由上一状态求出的隐藏层状态对应的概率值和由概率值转化而来的二进制状态，`vprob` 代表可见层概率值。

第 29 行：将可见层概率值作为下一次吉布斯采样的输入。

第 30 行：将可见层概率值与最后的隐藏层概率值进行点积运算，作为负向阶段的状态。

第 32、33 行：定义 TensorFlow 算子更新可见层与隐藏层之间的连接权值。

第 34、35 行：定义 TensorFlow 算子更新隐藏层偏置值。

第 36、37 行：定义 TensorFlow 算子更新可见层偏置值。

第 38 行：为了提高数值计算精度和稳定性，求出可见层概率值规整化的值，使其在 $1e-10$ 到正无穷之间。

第 39 行：为了提高数值计算精度和稳定性，求出 $1 - \text{可见层概率值规整化的值}$ ，使其在 $1e-10$ 到正无穷之间。

第 40~43 行：定义代价函数为原始输入信号与经过吉布斯采样后可见层概率值之间的交叉熵。公式如下：

$$\text{loss} = x \log v_{\text{prob}} + (1 - x) \log(1 - v_{\text{prob}})$$

第 44、45 行：最终的代价函数为交叉熵加上 L2 调整项（权值衰减），这里同时调整可见层与隐藏层之间的连接权值、可见层偏置值和隐藏层偏置值。

下面来看模型的训练方法，代码如下：

```
1 def train(self, mode=TRAIN_MODE_NEW, ckpt_file='work/rbm.ckpt'):
2     X_train, y_train, X_validation, y_validation, X_test, \
3         y_test, mnist = self.load_datasets()
4     with self.tf_graph.as_default():
5         self.build_model()
6         saver = tf.train.Saver()
7         with tf.Session() as sess:
8             sess.run(tf.global_variables_initializer())
```

```

9         for epoch in range(self.num_epochs):
10             np.random.shuffle(X_train)
11             batches = [_ for _ in self.gen_mini_batches(X_train,
12                                                         self.batch_size)]
13             total_batches = len(batches)
14             for idx, batch in enumerate(batches):
15                 cost_val, _, _, _ = sess.run([self.cost, self.w_upd8,
16                                               self.bh_upd8, self.bv_upd8], feed_dict={
17                     self.X: batch,
18                     self.hrand: np.random.rand(batch.shape[0],
19                                                 self.num_hidden),
20                     self.vrand: np.random.rand(batch.shape[0],
21                                                 batch.shape[1])})
22                 if (epoch*total_batches + idx) % 100 == 0:
23                     saver.save(sess, ckpt_file)
24                 print('{0}_{1}:cost={2}'.format(epoch, idx, cost_val))

```

第 2、3 行：载入 MNIST 数据集，得到训练样本集输入样本集、训练样本集标签集、验证样本集输入样本集、验证样本集标签集、测试样本集输入样本集、测试样本集标签集。

第 4 行：启动 TensorFlow 的 graph。

第 5 行：调用 build_model 方法创建受限玻尔兹曼机模型。

第 6 行：创建 TensorFlow 的 saver 对象，用于保存模型的参数和超参数。

第 7 行：启动 TensorFlow 会话。

第 8 行：初始化全局变量。

第 9 行：循环第 10~23 行操作，训练完整训练样本集指定遍数。

第 10 行：对训练样本集进行随机洗牌，打乱顺序。

第 11、12 行：以指定大小生成迷你批次列表 batches。

第 13 行：求出迷你批次总数。

第 14 行：循环第 15~23 行操作，处理每个迷你批次。

第 15~21 行：以本迷你批次样本集随机数初始化的隐藏层状态和随机数初始化的可见层状态为输入，求出模型的代价函数，网络会运行吉布斯采样并更新可见层和隐藏层之间的连接权值、可见层偏置值、隐藏层偏置值。

第 22、23 行：如果训练了 100 个迷你批次，则保存模型的参数和超参数到模型文件中。

第 24 行：打印当前的训练状态。

在训练方法中，我们用到了一些工具方法，下面分别进行介绍。

首先来看从可见层到隐藏层的采样方法，代码如下：

```

1 def sample_hidden_from_visible(self, vis_layer):
2     hprobs = tf.nn.sigmoid(tf.add(tf.matmul(vis_layer, self.W), self.bh_))
3     hstates = self.sample_prob(hprobs, self.hrand)
4     return hprobs, hstates
5
6 def sample_prob(self, probs, rand):
7     return tf.nn.relu(tf.sign(probs - rand))

```

第 2 行：由可见层状态求出隐藏层概率值，公式为：

$$h = \frac{1}{1 + e^{-(Wv+bh)}}$$

第 3 行：调用 `sample_prob` 方法，通过概率值求出二进制值表示的状态。

第 4 行：返回隐藏层的概率值和二进制值表示的状态。

第 6、7 行：定义 `sample_prob` 方法，将隐藏层节点的概率值与生成的 0~1 的随机值进行比较，如果隐藏层节点的概率值大就取 1，否则取 0。

下面来看由隐藏层采样可见层，代码如下：

```
1 def sample_visible_from_hidden(self, hidden, n_features):
2     visible_activation = tf.add(
3         tf.matmul(hidden, tf.transpose(self.W)),
4         self.bv_
5     )
6     if self.visible_unit_type == 'bin':
7         vprobs = tf.nn.sigmoid(visible_activation)
8     elif self.visible_unit_type == 'gauss':
9         vprobs = tf.truncated_normal(
10             (1, n_features), mean=visible_activation, stddev=self.stddev)
11     else:
12         vprobs = None
13     return vprobs
```

第 2~5 行：计算可见层的输入值。

第 6、7 行：如果可见层是二进制格式，采用 `Sigmoid` 函数计算概率值。

第 8、9 行：如果可见层为高斯型神经元，则采用以可见层输入值为均值、标准差为 0.1 的正态分布随机数作为可见层的概率值。

第 11、12 行：如果既不是二进制也不是高斯型，则概率值返回空。

第 13 行：返回可见层概率值。

下面来看吉布斯采样方法，代码如下：

```
1 def gibbs_sampling_step(self, visible, n_features):
2     hprobs, hstates = self.sample_hidden_from_visible(visible)
3     vprobs = self.sample_visible_from_hidden(hprobs, n_features)
4     hprobs1, hstates1 = self.sample_hidden_from_visible(vprobs)
5     return hprobs, hstates, vprobs, hprobs1, hstates1
```

第 2 行：由可见层采样隐藏层，得到隐藏层的概率值和状态。

第 3 行：由隐藏层采样可见层，得到可见层的概率值。

第 4 行：重新由可见层采样隐藏层，得到隐藏层新的概率值和状态。

第 5 行：返回第一次由可见层采样隐藏层时隐藏层的概率值和状态、可见层概率值，以及第二次由可见层采样隐藏层时隐藏层的概率值和状态。

下面来看求正向阶段可见层状态的方法，代码如下：

```
1 def compute_positive_association(self, visible,
2                                 hidden_probs, hidden_states):
3     if self.visible_unit_type == 'bin':
4         positive = tf.matmul(tf.transpose(visible), hidden_states)
5     elif self.visible_unit_type == 'gauss':
```

```

6         positive = tf.matmul(tf.transpose(visible), hidden_probs)
7     else:
8         positive = None
9     return positive

```

第 3、4 行：如果可见层为二进制类型，则使用隐藏层状态与可见层状态的乘积（形状为可见层维度×隐藏层维度）来表示。

第 5、6 行：如果可见层为高斯型，则使用隐藏层概率值与可见层状态的乘积（形状为可见层维度×隐藏层维度）来表示。

第 7、8 行：否则返回空。

第 9 行：返回计算出的值。

产生迷你批次的方法的代码如下：

```

1     def gen_mini_batches(self, X, batch_size):
2         X = np.array(X)
3         for i in range(0, X.shape[0], batch_size):
4             yield X[i:i + batch_size]

```

这段代码先将样本变为 numpy 的数组，将其分割为 batch_size 大小的子数组，用 yield 函数将其作为一个序列。

运行训练方法会产生如图 10.4 所示的输出结果。

```

49_405:cost=0.22784185409545898
49_406:cost=0.2349613606929779
49_407:cost=0.22663630545139313
49_408:cost=0.23190124332904816
49_409:cost=0.23230254650115967
49_410:cost=0.2287660837173462
49_411:cost=0.23792517185211182
49_412:cost=0.2231128215789795
49_413:cost=0.22912217676639557
49_414:cost=0.23067979514598846
49_415:cost=0.22820572555065155
49_416:cost=0.23266898095607758
49_417:cost=0.2282627671957016
49_418:cost=0.23120498657226562
49_419:cost=0.23171883821487427
49_420:cost=0.22501717507839203
49_421:cost=0.2232668101787567
49_422:cost=0.22564196586608887
49_423:cost=0.23439495265483856
49_424:cost=0.22752924263477325
49_425:cost=0.2352072149515152
49_426:cost=0.2290208488702774
49_427:cost=0.23216310143470764
49_428:cost=0.22554631531238556
49_429:cost=0.23142017424106598

```

图 10.4 受限玻尔兹曼机训练结果输出

受限玻尔兹曼机是一种生成式网络，它的一个特性就是可以通过对可见层的采样得到网络生成的样本。为了让大家对训练好的受限玻尔兹曼机有一个感性的认识，下面将对受限玻尔兹曼机进行采样，生成人工样本，代码如下：

```

1     def run(self, ckpt_file='work/rbm.ckpt'):
2         raw = np.random.normal(0, 0.1, self.n)
3         X_train, y_train, X_validation, y_validation, X_test, \
4             y_test, mnist = self.load_datasets()
5         raw = X_train[3098]
6         batch = np.reshape(raw, [1, self.n])
7         with self.tf_graph.as_default():
8             self.build_model()

```

```

9     saver = tf.train.Saver()
10    with tf.Session() as sess:
11        sess.run(tf.global_variables_initializer())
12        saver.restore(sess, ckpt_file)
13        vprob = sess.run(self.vprob, feed_dict={
14            self.X: batch,
15            self.hrand: np.random.rand(batch.shape[0],
16                                       self.num_hidden),
17            self.vrand: np.random.rand(batch.shape[0],
18                                       batch.shape[1])})
19        print(vprob)
20        plt.figure(1)
21        plt.subplot(121)
22        img0 = np.reshape(raw, [28, 28])
23        plt.imshow(img0, cmap='gray')
24        plt.subplot(122)
25        img = np.reshape(vprob, [28, 28])
26        plt.imshow(img, cmap='gray')
27        plt.show()

```

第 2 行：生成一个随机样本。

第 3、4 行：载入 MNIST 数据集，得到训练样本集输入样本集、训练样本集标签集、验证样本集输入样本集、验证样本集标签集、测试样本集输入样本集、测试样本集标签集。

第 5 行：取出测试样本集的第 3099 号样本。

第 6 行：将该样本组成只有一个样本的迷你批次。

第 7 行：打开 TensorFlow 的计算图。

第 8 行：调用 `build_model` 方法创建模型。

第 9 行：创建 TensorFlow 的 `saver` 对象，用于恢复模型的参数和超参数。

第 10 行：启动 TensorFlow 会话。

第 11 行：初始化全局变量。

第 12 行：从训练阶段保存的模型文件中恢复参数和超参数。

第 13~18 行：在当前迷你批次（由当前一个样本组成的迷你批次），利用 0~1 的随机数初始化隐藏层和可见层，调用 TensorFlow 计算经过吉布斯采样后可见层的概率值。

第 19 行：打印可见层概率值信息。

第 20 行：初始化图形绘制库。

第 21 行：在 1 行 2 列的网格的第 1 行第 1 列中绘制。

第 22 行：将输入信号转换为 28×28 的黑白图像。

第 23 行：显示原始图像。

第 24 行：在第 1 行第 2 列进行图形绘制。

第 25 行：将可见层概率值变为 28×28 的黑白图像格式。

第 26 行：绘制可见层重构的图像。

第 27 行：显示图像。

运行上面的程序，会产生如图 10.5 所示的输出。

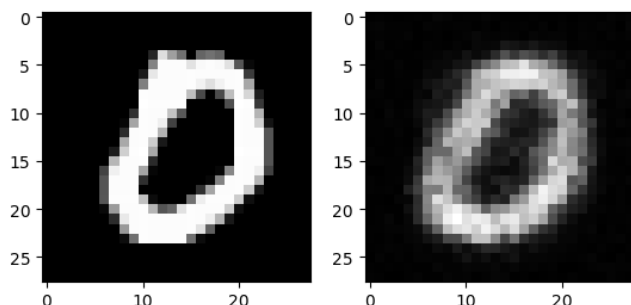


图 10.5 受限玻尔兹曼机采样生成的样本

由图可以看出，受限玻尔兹曼机确实可以生成比较真实的图片，说明我们的实现是正确的。需要注意的是，我们只是给大家讲了一个例子，在训练时只对训练样本集训练了 50 次，所以我们的网络并没有达到最佳性能。读者可以通过增加 `self.num_epochs` 的值，调整隐藏层神经元数，再通过加入 `Early Stopping` 调整机制等，来提高模型采样时得到样本的精度。建议读者自行调整参数和超参数，看看会出现什么情况，以加深对模型的理解。

在实际应用中，很少单独使用受限玻尔兹曼机，其通常作为深度信念网络的预训练层使用，我们将在下一章向大家做详细介绍。

10.3 受限玻尔兹曼机 Theano 实现

明白了受限玻尔兹曼机的基本原理之后，来看看在 Theano 框架下实现受限玻尔兹曼机的方法。

首先定义一个受限玻尔兹曼机类，构造函数如下：

```
1 from __future__ import print_function
2 import timeit
3 try:
4     import PIL.Image as Image
5 except ImportError:
6     import Image
7 import numpy
8 import theano
9 import theano.tensor as T
10 import os
11 from theano.sandbox.rng_mrg import MRG_RandomStreams as RandomStreams
12 from utils import tile_raster_images
13 from logistic_sgd import load_data
14
15 class RBM(object):
16     def __init__(
17         self,
18         input=None,
19         n_visible=784,
20         n_hidden=500,
21         W=None,
22         hbias=None,
```

```

23     vbias=None,
24     numpy_rng=None,
25     theano_rng=None
26 ):
27     """
28     RBM类构造函数，定义PCD算法所需参数和超参数值。可以工作于单独模式
29     也可以组成深度信息网络（DBN）。
30     参数：
31     - input：如果是单独的RBM，则此参数为None，但是如果是深度信念网络
32       一层的话，则其将是Theano变量，表示下一层的输出
33     - n_visible：可见层神经元数量
34     - n_hidden：隐藏层神经元数量
35     - W：可见层与隐藏层间的连接权值矩阵
36     - hbias：隐藏层神经元偏移量
37     - vbias：可见层神经元偏移量
38     """
39
40     self.n_visible = n_visible
41     self.n_hidden = n_hidden
42
43     if numpy_rng is None:
44         # create a number generator
45         numpy_rng = numpy.random.RandomState(1234)
46
47     if theano_rng is None:
48         theano_rng = RandomStreams(numpy_rng.randint(2 ** 30))
49
50     if W is None:
51         # W is initialized with 'initial_W' which is uniformly
52         # sampled from -4*sqrt(6./(n_visible+n_hidden)) and
53         # 4*sqrt(6./(n_hidden+n_visible)) the output of uniform if
54         # converted using asarray to dtype theano.config.floatX so
55         # that the code is runnable on GPU
56         initial_W = numpy.asarray(
57             numpy_rng.uniform(
58                 low=-4 * numpy.sqrt(6. / (n_hidden + n_visible)),
59                 high=4 * numpy.sqrt(6. / (n_hidden + n_visible)),
60                 size=(n_visible, n_hidden)
61             ),
62             dtype=theano.config.floatX
63         )
64         # theano shared variables for weights and biases
65         W = theano.shared(value=initial_W, name='W', borrow=True)
66
67     if hbias is None:
68         # create shared variable for hidden units bias
69         hbias = theano.shared(
70             value=numpy.zeros(
71                 n_hidden,
72                 dtype=theano.config.floatX
73             ),
74             name='hbias',
75             borrow=True
76         )
77
78     if vbias is None:
79         # create shared variable for visible units bias
80         vbias = theano.shared(
81             value=numpy.zeros(
82                 n_visible,
83                 dtype=theano.config.floatX
84             ),
85             name='vbias',
86             borrow=True
87         )
88
89     # initialize input layer for standalone RBM or layer0 of DBN
90     self.input = input
91     if not input:
92         self.input = T.matrix('input')
93
94     self.W = W
95     self.hbias = hbias
96     self.vbias = vbias
97     self.theano_rng = theano_rng
98     # **** WARNING: It is not a good idea to put things in this list

```



```

99      # other than shared variables created in this function.
100     self.params = [self.W, self.hbias, self.vbias]

```

第 40 行：定义属性 `n_visible`，表示可见层神经元数量。

第 41 行：定义属性 `n_hidden`，表示隐藏层神经元数量。

第 43~45 行：如果没有 `numpy` 随机数生成引擎，则初始化 `numpy_rng`。

第 47、48 行：如果没有定义随机数生成引擎 `theano_rng`，则进行初始化。

第 50~65 行：如果连接权值矩阵 `W` 的参数没有设置，则进行设置，取以下范围的随机数：

$$\left[-4 \times \sqrt{\frac{6.0}{n_{\text{visible}} + n_{\text{hidden}}}}, 4 \times \sqrt{\frac{6.0}{n_{\text{visible}} + n_{\text{hidden}}}} \right]$$

我们将其设置为数组，指定其类型为 `floatX`，这样就可以无缝应用于 GPU 中，由机器自动来选择，当然也可以在运行时通过参数 `THEANO_FLAGS="floatX=float32"` 来指定。生成随机数数组之后，将 `W` 定义为 Theano 变量。

第 67~76 行：如果隐藏层神经元偏移量 `hbias` 参数没有设置，则将全零的数组赋给 `hbias`，并将其定义为 Theano 变量。

第 78~87 行：如果可见层神经元偏移量 `vbias` 参数没有设置，则将全零的数组赋给 `vbias`，并将其定义为 Theano 变量。

第 90 行：定义 `input` 属性，并由参数 `input` 来初始化。

第 91、92 行：如果是单独运行或是深度信念网络的第一层，则对输入层进行初始化。

第 94 行：定义属性 `W` 为连接权值矩阵 `W`。

第 95 行：定义属性 `hbias` 为隐藏层神经元偏移量参数 `hbias`。

第 96 行：定义属性 `vbias` 为可见层神经元偏移量参数 `vbias`。

第 97 行：定义属性 `theano_rng`，用于随机数生成。

第 100 行：定义模型的参数为连接权值矩阵、隐藏层神经元偏移量、可见层神经元偏移量。

研究人员受物理系统启发，定义了自由能，在受限玻尔兹曼机中，其定义为：

$$F(\mathbf{v}) = -\log \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})} \quad (32)$$

则求自由能函数的代码如下：

```

102     def free_energy(self, v_sample):
103         ''' 计算自由能 '''
104         wx_b = T.dot(v_sample, self.W) + self.hbias
105         vbias_term = T.dot(v_sample, self.vbias)
106         hidden_term = T.sum(T.log(1 + T.exp(wx_b)), axis=1)
107         return -hidden_term - vbias_term

```

按照理论部分的符号的意义，推导这部分程序的实现。首先由能量函数定义：

$$E(\mathbf{v}, \mathbf{h} | \boldsymbol{\theta}) = -\sum_{i=1}^n \sum_{j=1}^m v_j W_{ij} h_i - \sum_{j=1}^m b_j v_j - \sum_{i=1}^n c_i h_i \quad (33)$$

可以将第一项和第三项进行合并：

$$\begin{aligned}
 E(\mathbf{v}, \mathbf{h} | \boldsymbol{\theta}) &= - \sum_{i=1}^n \sum_{j=1}^m v_j W_{ij} h_i - \sum_{j=1}^m b_j v_j - \sum_{i=1}^n c_i h_i \\
 &= - \sum_{i=1}^n \sum_{j=1}^m (v_j W_{ij} h_i + c_i h_i) - \sum_{j=1}^m b_j v_j \\
 &= - \sum_{i=1}^n \sum_{j=1}^m (v_j W_{ij} + c_i) h_i - \sum_{j=1}^m b_j v_j \\
 &= - \sum_{i=1}^n h_i \sum_{j=1}^m (v_j W_{ij} + c_i) - \sum_{j=1}^m b_j v_j
 \end{aligned} \tag{34}$$

则自由能定义为：

$$F(\mathbf{v}) = - \sum_{j=1}^m b_j v_j - \sum_i \log \sum_{h_i} e^{h_i (W_i \mathbf{v} + c_i)} \tag{35}$$

第 104 行：计算 $\sum_{j=1}^m (v_j W_{ij} + c_i)$ 的值。

第 105 行：计算 $\sum_{j=1}^m b_j v_j$ 的值。

第 106 行：计算式 (35) 的第二项。

第 107 行：返回式 (35) 的完整值。

下面来看信号从可见层传输到隐藏层，代码如下：

```

109 def propup(self, vis):
110     """
111     将可见层信息传输到隐藏层
112     参数
113     - vis : 可见层状态
114     返回值
115     - pre : 隐藏层接收到的线性输入 z=Wv+c
116     - out : sigmoid(z)
117     注意：我们在这里不仅返回隐藏层的输出值，同时传回了隐藏层的线性
118     输入值，因为Theano需要这样来进行优化计算。我们将在
119     get_reconstruction_cost函数具体讲述原因。
120     """
121     pre_sigmoid_activation = T.dot(vis, self.W) + self.hbias
122     return [pre_sigmoid_activation, T.nnet.sigmoid(pre_sigmoid_activation)]
123

```

这个函数实际上对应于理论部分的公式：

$$\begin{aligned}
 z_i &= \sum_{j=1}^m W_{ij} v_j + c_i \\
 h_i &= \frac{1}{1 + e^{-z_i}}
 \end{aligned}$$

程序分别返回这两个值。

下面来看已知可见层输入，对隐藏层进行采样的函数实现，代码如下：

```

124 def sample_h_given_v(self, v0_sample):
125     ''' 对隐藏层进行采样，将概率值转化为0、1值 '''
126     # compute the activation of the hidden units given a sample of
127     # the visibles
128     pre_sigmoid_h1, h1_mean = self.propup(v0_sample)
129     # get a sample of the hiddens given their activation
130     # Note that theano_rng.binomial returns a symbolic sample of dtype
131     # int64 by default. If we want to keep our computations in floatX
132     # for the GPU we need to specify to return the dtype floatX
133     h1_sample = self.theano_rng.binomial(size=h1_mean.shape,
134                                         n=1, p=h1_mean,
135                                         dtype=theano.config.floatX)
136     return [pre_sigmoid_h1, h1_mean, h1_sample]
137

```

第 128 行：调用 `self.propup` 将信号传播到隐藏层，得到隐藏层输出。

第 133~135 行：调用 `binomial` 函数，对隐藏层进行一次采样。这部分代码不太好理解，给大家举一个简单的例子：

```

1 import theano
2 import numpy
3 from theano.sandbox.rng_mrg import MRG_RandomStreams as RandomStreams
4
5 numpy_rng = numpy.random.RandomState(1234)
6 rng = RandomStreams(numpy_rng.randint(2 ** 30))
7 h1_mean = numpy.asarray([0.1, 0.3, 0.8, 0.86, 0.9])
8 h1_sample = rng.binomial(size=h1_mean.shape, n=1, p=h1_mean)
9
10 print(h1_sample.eval())

```

运行结果为：

```
[0 0 1 1 1]
```

从这个例子可以看出，对隐藏层的输出状态，如上例中的`[0.1, 0.3, 0.8, 0.86, 0.9]`，可以理解为概率值，采样程序将大的值采样为 1，将小的值采样为 0。

第 136 行：将隐藏层线性输入值、输出值、采样值作为返回值返回。

当信号传输到隐藏层后，会再次传输到可见层，代码如下：

```

138 def propdown(self, hid):
139     '''
140     将隐藏层输出值向下传输到可见层
141     参数
142     - hid : 隐藏层输出值
143     返回值
144     - pre_sigmoid_activation : 输入线性和
145     - activation : 可见层输出值
146     '''
147     pre_sigmoid_activation = T.dot(hid, self.W.T) + self.vbias
148     return [pre_sigmoid_activation, T.nnet.sigmoid(pre_sigmoid_activation)]
149

```

根据前面的理论部分介绍，这段代码对应的公式为：

$$z_j^{(1)(2)} = \sum_{i=1}^n W_{ij}^{(1,2)} a_i^{(2)(2)} = \sum_{i=1}^n W_{ij} h_i^{(2)}$$

$$v_j^{(2)} = a_j^{(1)(2)} = \sigma(z_j^{(1)(2)}) = \frac{1}{1 + e^{-z_j^{(1)(2)}}}$$

式中， z 表示线性输入和； a 表示神经元输出，下标代表神经元编号，上标第一项代表层数，

第二项代表时刻； W 的上标表示 1、2 层间的连接权值矩阵。

下面对可见层进行采样，代码如下：

```
150 def sample_v_given_h(self, h0_sample):
151     ''' 对可见层采样 '''
152     # 计算可见层线性输入值和输出值
153     pre_sigmoid_v1, v1_mean = self.propdown(h0_sample)
154     # get a sample of the visible given their activation
155     # Note that theano_rng.binomial returns a symbolic sample of dtype
156     # int64 by default. If we want to keep our computations in floatX
157     # for the GPU we need to specify to return the dtype floatX
158     v1_sample = self.theano_rng.binomial(size=v1_mean.shape,
159                                         n=1, p=v1_mean,
160                                         dtype=self.config.floatX)
161     return [pre_sigmoid_v1, v1_mean, v1_sample]
162
```

第 153 行：调用 `self.propdown` 求出可见层的线性输入值和输出值。

第 158~160 行：调用 `binomial`，根据可见层神经元输出值，进行 0、1 的二值采样。

下面来看从隐藏层到可见层，再从可见层到隐藏层的吉布斯采样，代码如下：

```
163 def gibbs_hvh(self, h0_sample):
164     '''
165     给定隐藏层状态，先将其传输到可见层，并对可见层进行采样，然后将可见层
166     信号传输到隐藏层，并对隐藏层进行采样
167     '''
168     pre_sigmoid_v1, v1_mean, v1_sample = self.sample_v_given_h(h0_sample)
169     pre_sigmoid_h1, h1_mean, h1_sample = self.sample_h_given_v(v1_sample)
170     return [pre_sigmoid_v1, v1_mean, v1_sample,
171           pre_sigmoid_h1, h1_mean, h1_sample]
172
```

给定隐藏层状态，先将信号传输到可见层，并对可见层进行采样，再将该信号传输回隐藏层，并对隐藏层进行采样。

下面来看从可见层到隐藏层，再从隐藏层到可见层的吉布斯采样，代码如下：

```
173 def gibbs_vhv(self, v0_sample):
174     '''
175     给定可见层状态，先将其传输到隐藏层，并对隐藏层进行采样，然后将隐藏层
176     信号传输到可见层，并对可见层进行采样
177     '''
178     pre_sigmoid_h1, h1_mean, h1_sample = self.sample_h_given_v(v0_sample)
179     pre_sigmoid_v1, v1_mean, v1_sample = self.sample_v_given_h(h1_sample)
180     return [pre_sigmoid_h1, h1_mean, h1_sample,
181           pre_sigmoid_v1, v1_mean, v1_sample]
182
```

给定可见层状态，先将信号传输到隐藏层，并对隐藏层进行采样，然后将该信号传输回可见层，并对可见层进行采样。

```
183 def get_cost_updates(self, lr=0.1, persistent=None, k=1):
184     """
185     实现CD-K、PCD-K算法的一步
186     参数
187     - lr : 学习率
188     - persistent : 为空时为CD-K算法，否则为Theano共享变量，保存吉布斯
189                   (Gibbs) 采样之前的状态，格式为 (迷你批次中样本数，
190                   隐藏层神经元)
191     - k : CD-K的K值
192     返回值
193     代价函数值和连接权值w、可见层偏移量、隐藏层偏移量、
194     persistent (如果是PCD-K算法时)
195     """
196     # compute positive phase
197     pre_sigmoid_ph, ph_mean, ph_sample = self.sample_h_given_v(self.input)
198     # 初始化隐藏层：
199     # 对CD-K算法：直接使用上面计算的隐藏层采样值
200
```

```

201 # 对PCD-K算法：使用参数persistent传递过来的参数
202 if persistent is None:
203     chain_start = ph_sample
204 else:
205     chain_start = persistent
206 # perform actual negative phase
207 # in order to implement CD-k/PCD-k we need to scan over the
208 # function that implements one gibbs step k times.
209 # Read Theano tutorial on scan for more information :
210 # http://deeplearning.net/software/theano/library/scan.html
211 # the scan will return the entire Gibbs chain
212 (
213     [
214         pre_sigmoid_nvs,
215         nv_means,
216         nv_samples,
217         pre_sigmoid_nhs,
218         nh_means,
219         nh_samples
220     ],
221     updates
222 ) = theano.scan(
223     self.gibbs_hvh,
224     # the None are place holders, saying that
225     # chain_start is the initial state corresponding to the
226     # 6th output
227     outputs_info=[None, None, None, None, None, chain_start],
228     n_steps=k,
229     name="gibbs_hvh"
230 )
231 # determine gradients on RBM parameters
232 # note that we only need the sample at the end of the chain
233 chain_end = nv_samples[-1]
234
235 cost = T.mean(self.free_energy(self.input)) - T.mean(
236     self.free_energy(chain_end))
237 # We must not compute the gradient through the gibbs sampling
238 gparams = T.grad(cost, self.params, consider_constant=[chain_end])
239 # constructs the update dictionary
240 for gparam, param in zip(gparams, self.params):
241     # make sure that the learning rate is of the right dtype
242     updates[param] = param - gparam * T.cast(
243         lr,
244         dtype=theano.config.floatX
245     )
246 if persistent:
247     # Note that this works only if persistent is a shared variable
248     updates[persistent] = nh_samples[-1]
249     # pseudo-likelihood is a better proxy for PCD
250     monitoring_cost = self.get_pseudo_likelihood_cost(updates)
251 else:
252     # reconstruction cross-entropy is a better proxy for CD
253     monitoring_cost = self.get_reconstruction_cost(updates,
254         pre_sigmoid_nvs[-1])
255
256 return monitoring_cost, updates

```

第 197 行：先将输入信号加载到可见层，再将信号传输到隐藏层，并对隐藏层进行采样，得到隐藏层线性输入和、隐藏层神经元输出值、隐藏层神经元状态（0、1）。

第 202~205 行：如果参数 `persistent` 为空，表示为 CD-K 算法，此时马可夫链开始设置为刚求出来的隐藏层状态；如果参数 `persistent` 不为空，则将马可夫链初始状态设置为参数 `persistent` 中的状态。

第 212~230 行：调用 Theano 中的 `scan` 函数，执行吉布斯采样，包括各步的可见层、隐藏层状态和更新值。Theano 中的 `scan` 函数主要用于执行循环操作。

第 233 行：将采样最后一步的可见层状态设置为马可夫链结束点状态。

第 235~236 行：将马可夫链开始时的平均自由能与结束时的平均自由能相减，定义为代价函数值。

第 238 行：求代价函数对模型各参数的导数。

第 240~245 行：根据模型各参数的导数，再结合学习算法（`param=param-lr*gparam`），求出各参数的更新值，并保存在 `updates` 字典中。在学习算法中只用了梯度下降算法，没有加动量项、L2 调整项等。

第 246 行：判断是否为 PCD-K 算法，采用不同的算法求代价函数监控值。

第 248~250 行：如果是 PCD-K 算法，将更新字典 `updates` 中的 `persistent` 设置为采样最后一步隐藏层状态，调用 `get_pseudo_likelihood_cost` 函数计算代价函数监控值。

第 253、254 行：如果是 CD-K 算法，调用 `get_reconstruction_cost` 函数求代价函数监控值。

第 256 行：返回代价函数监控值和参数更新字典 `updates`。

受限玻尔兹曼机非常难以进行训练，因为存在归一化因子 Z ，在训练过程中，我们不能很好地估计对数似然函数 $\log(P(x))$ ，所以不能直接找到有效的参数来表示训练进程。

虽然有多种方法来进行训练进程监控，但是实际中使用最多的还是似然代理法。在采用 PCD-K 算法训练受限玻尔兹曼机时，可以采用伪似然函数（ PL ）作为真实似然函数的代理。伪似然函数计算比较简单，因为它假设所有点都是独立的。公式如下：

$$PL(x) = \prod_i P(x_i | x_{-i})$$

$$\log PL(x) = \sum_i \log P(x_i | x_{-i})$$

式中， x_{-i} 为除去下标为 i 的神经元的所有神经元。所以对数伪似然函数就是所有神经元对数概率值（给定其他神经元状态的情况下）之和。

在 MNIST 手写数字识别应用中，上式需要计算 784（ 28×28 ）次，计算量还是比较大的，因此通常使用对数概率随机逼近方法，公式如下：

$$g = N \cdot \log P(x_i | x_{-i}), \quad i \sim U(0, N)$$

$$E[g] = \log PL(x)$$

式中， E 代表期望值。

设可见层共有 N 个神经元，因为都是二值（仅可取 0、1 值）神经元，用 \tilde{x}_i 来表示第 i 个神经元发生了状态转换，即从 1 转换到 0，或者从 0 转换到 1，所以伪似然函数近似逼近可以表示为：

$$\log PL(x) \approx N \cdot \log \frac{e^{-FE(x)}}{e^{-FE(x)} + e^{-FE(\tilde{x}_i)}} \approx N \cdot \log [\text{sigmoid}(FE(\tilde{x}_i) - FE(x))]$$

式中， FE 代表自由能函数。这就是我们最终求代价函数近似值的方法，代码实现如下：

```
258 def get_pseudo_likelihood_cost(self, updates):
259     """ 求在PCD-K算法中用的伪似然函数的随机逼近近似值 """
260
261     # index of bit i in expression p(x_i | x_{-i})
262     bit_i_idx = theano.shared(value=0, name='bit_i_idx')
263
264     # binarize the input image by rounding to nearest integer
```

```

265     xi = T.round(self.input)
266
267     # calculate free energy for the given bit configuration
268     fe_xi = self.free_energy(xi)
269
270     # flip bit x_i of matrix xi and preserve all other bits x_{\i}
271     # Equivalent to xi[:,bit_i_idx] = 1-xi[:, bit_i_idx], but assigns
272     # the result to xi_flip, instead of working in place on xi.
273     xi_flip = T.set_subtensor(xi[:, bit_i_idx], 1 - xi[:, bit_i_idx])
274
275     # calculate free energy with bit flipped
276     fe_xi_flip = self.free_energy(xi_flip)
277
278     # equivalent to e^(-FE(x_i)) / (e^(-FE(x_i)) + e^(-FE(x_{\i})))
279     cost = T.mean(self.n_visible * T.log(T.nnet.sigmoid(fe_xi_flip -
280                                                         fe_xi)))
281
282     # increment bit_i_idx % number as part of updates
283     updates[bit_i_idx] = (bit_i_idx + 1) % self.n_visible
284
285     return cost

```

第 262 行：定义 Theano 变量 `bit_i_idx`，表示可见层神经元序号。

第 265 行：定义 `xi` 表示输入信号。

第 268 行：计算当前状态下的自由能 `fe_xi`。

第 273 行：对迷你批次所有样本中第 i 个可见层神经元转换状态。

第 276 行：计算转换状态后的自由能。

第 277 行：利用下面的公式计算代价函数近似值：

$$N \cdot \log \frac{e^{-FE(x)}}{e^{-FE(x)} + e^{-FE(\tilde{x}_i)}}$$

第 283 行：更新可见层神经元序号。

第 285 行：返回代价函数值。

这些代码是受限玻尔兹曼机采用 PCD-K 算法时，计算代价函数的过程。如果使用 CD-K 算法，则应使用 `get_reconstruction_cost` 来计算代价函数值。

在看具体代码实现之前，我们先来讲一下为什么信号在可见层和隐藏层传输时，不仅要保存输出值，而且要保存线性输入值。为了解答这个问题，我们需要先讲解 Theano 的具体实现机制。当我们定义一个 Theano 函数后，Theano 会对该函数进行编译。通过输入参数传给 Theano 的计算图，会在速度和稳定性方面进行优化，这会更改原来定义的计算图，其中就包括将 $\log(\text{sigmoid}(x))$ 计算转化为 softmax 函数。例如我们在进行交叉熵计算时就需要这么做，因为当 x 值大于 30 时， $\text{sigmoid}(x)$ 的值将为 1，当 x 值小于 -30 时， $\text{sigmoid}(x)$ 的值将为 0，这时 $\log(\text{sigmoid}(x))$ 的值就是负无穷或非数值，但是使用 softmax 函数就不会出现这一现象。这个特性通常会满足我们的实际需要，但是在代价函数进行吉布斯采样时，使用 Theano 中的 `scan` 函数进行 K 步循环采样，这时 Theano 优化程序就会认为计算是 $\log(\text{scan}(x))$ 形式，就不进行优化了，就会出现负无穷或非数值的现象。因此可以将线性输入和放在 `scan` 函数的外面进行 $\log(\text{sigmoid}(x))$ 计算，这样 Theano 就可以进行优化工作了。

下面来看 `get_reconstruction_cost` 函数，代码如下：

```

287 def get_reconstruction_cost(self, updates, pre_sigmoid_nv):
288     """
289     在使用CD-K算法时，采用本函数计算代价函数值，此时用信息交叉熵的方法
290     作为代价函数值
291     """
292     cross_entropy = T.mean(
293         T.sum(
294             self.input * T.log(T.nnet.sigmoid(pre_sigmoid_nv)) +
295             (1 - self.input) * T.log(1 - T.nnet.sigmoid(pre_sigmoid_nv)),
296             axis=1
297         )
298     )
299     return cross_entropy

```

在完成了所有准备工作之后，就可以使用受限玻尔兹曼机了。下面我们来看其训练过程，代码如下：

```

302 def test_rbm(learning_rate=0.1, training_epochs=15,
303              dataset='mnist.pkl.gz', batch_size=20,
304              n_chains=20, n_samples=10, output_folder='rbm_plots',
305              n_hidden=500):
306     """
307     以MNIST手写字母识别为目标，训练受限玻尔兹曼机（RBM）
308     参数
309     - learning_rate：学习率
310     - training_epochs：训练样本遍历次数
311     - dataset：数据集
312     - batch_size：迷你批次中样本数
313     - n_chains：在采样过程中，吉布斯采样马可夫链并行数量
314     - n_samples：在每个马可夫链中需绘制的采样点数量
315     - output_folder：绘制权值图像的输出目录
316     - n_hidden：隐藏层神经元数量
317     """
318     datasets = load_data(dataset)
319     train_set_x, train_set_y = datasets[0]
320     test_set_x, test_set_y = datasets[2]
321
322     # compute number of minibatches for training, validation and testing
323     n_train_batches = train_set_x.get_value(borrow=True).shape[0] // batch_size
324
325     # allocate symbolic variables for the data
326     index = T.lscalar() # index to a [mini]batch
327     x = T.matrix('x') # the data is presented as rasterized images
328
329     rng = numpy.random.RandomState(123)
330     theano_rng = RandomStreams(rng.randint(2 ** 30))
331
332     # initialize storage for the persistent chain (state = hidden
333     # layer of chain)
334     persistent_chain = theano.shared(numpy.zeros((batch_size, n_hidden),
335                                                  dtype=theano.config.floatX),
336                                     borrow=True)
337
338     # construct the RBM class
339     rbm = RBM(input=x, n_visible=28 * 28,
340              n_hidden=n_hidden, numpy_rng=rng, theano_rng=theano_rng)
341
342     # get the cost and the gradient corresponding to one step of CD-15
343     cost, updates = rbm.get_cost_updates(lr=learning_rate,
344                                         persistent=persistent_chain, k=15)
345
346     #####
347     # Training the RBM #
348     #####
349     if not os.path.isdir(output_folder):
350         os.makedirs(output_folder)
351     os.chdir(output_folder)
352
353     # it is ok for a theano function to have no output
354     # the purpose of train_rbm is solely to update the RBM parameters
355     train_rbm = theano.function(
356         [index],
357         cost,
358         updates=updates,

```



```

359     givens={
360         x: train_set_x[index * batch_size: (index + 1) * batch_size]
361     },
362     name='train_rbm'
363 )
364
365 plotting_time = 0.
366 start_time = timeit.default_timer()
367
368 # go through training epochs
369 for epoch in range(training_epochs):
370
371     # go through the training set
372     mean_cost = []
373     for batch_index in range(n_train_batches):
374         mean_cost += [train_rbm(batch_index)]
375
376     print('Training epoch %d, cost is ' % epoch, numpy.mean(mean_cost))
377
378     # Plot filters after each training epoch
379     plotting_start = timeit.default_timer()
380     # Construct image from the weight matrix
381     image = Image.fromarray(
382         tile_raster_images(
383             X=rbm.W.get_value(borrow=True).T,
384             img_shape=(28, 28),
385             tile_shape=(10, 10),
386             tile_spacing=(1, 1)
387         )
388     )
389     image.save('filters_at_epoch%i.png' % epoch)
390     plotting_stop = timeit.default_timer()
391     plotting_time += (plotting_stop - plotting_start)
392
393 end_time = timeit.default_timer()
394
395 pretraining_time = (end_time - start_time) - plotting_time
396
397 print('Training took %f minutes' % (pretraining_time / 60.))
398 #####
399 # Sampling from the RBM #
400 #####
401 # find out the number of test samples
402 number_of_test_samples = test_set_x.get_value(borrow=True).shape[0]
403
404 # pick random test examples, with which to initialize the persistent chain
405 test_idx = rng.randint(number_of_test_samples - n_chains)
406 persistent_vis_chain = theano.shared(
407     numpy.asarray(
408         test_set_x.get_value(borrow=True)[test_idx:test_idx + n_chains],
409         dtype=theano.config.floatX
410     )
411 )
412 plot_every = 1000
413 # define one step of Gibbs sampling (mf = mean-field) define a
414 # function that does 'plot_every' steps before returning the
415 # sample for plotting
416 (
417     [
418         presig_hids,
419         hid_mfs,
420         hid_samples,
421         presig_vis,
422         vis_mfs,
423         vis_samples
424     ],
425     updates
426 ) = theano.scan(
427     rbm.gibbs_vhv,
428     outputs_info=[None, None, None, None, None, persistent_vis_chain],
429     n_steps=plot_every,
430     name="gibbs_vhv"
431 )
432

```

```

433 # add to updates the shared variable that takes care of our persistent
434 # chain :.
435 updates.update({persistent_vis_chain: vis_samples[-1]})
436 # construct the function that implements our persistent chain.
437 # we generate the "mean field" activations for plotting and the actual
438 # samples for reinitializing the state of our persistent chain
439 sample_fn = theano.function(
440     [],
441     [
442         vis_mfs[-1],
443         vis_samples[-1]
444     ],
445     updates=updates,
446     name='sample_fn'
447 )
448 # create a space to store the image for plotting ( we need to leave
449 # room for the tile_spacing as well)
450 image_data = numpy.zeros(
451     (29 * n_samples + 1, 29 * n_chains - 1),
452     dtype='uint8'
453 )
454 for idx in range(n_samples):
455     # generate 'plot_every' intermediate samples that we discard,
456     # because successive samples in the chain are too correlated
457     vis_mf, vis_sample = sample_fn()
458     print('... plotting sample %d' % idx)
459     image_data[29 * idx:29 * idx + 28, :] = tile_raster_images(
460         X=vis_mf,
461         img_shape=(28, 28),
462         tile_shape=(1, n_chains),
463         tile_spacing=(1, 1)
464     )
465 # construct image
466 image = Image.fromarray(image_data)
467 image.save('samples.png')
468 os.chdir('../')

```

第 318 行：载入 MNIST 手写数字识别数据集。

第 319 行：取出训练样本集输入集和输出标签集。

第 320 行：取出测试样本集输入集和输出标签集。由于受限玻尔兹曼机属于非监督学习方法，所以不需要输出标签集。

第 323 行：求出迷你批次数量 `n_train_batches`。

第 326 行：定义迷你批次索引号。

第 327 行：定义设计矩阵，其有 `batch_size` 行，每行有 784 (28×28) 列，即对应一个样本。

第 329 行：定义随机数生成引擎 `rng`。

第 330 行：定义 Theano 随机数生成引擎 `theano_rng`。

第 334~336 行：由于实践证明 PCD-K 算法比 CD-K 算法的效率要高，所以在本例中使用 PCD-K 算法，`persistent_chain` 取为隐藏层状态，其为 Theano 共享变量，为二维数组，初始时元素均为零，形状为（迷你批次中的样本数，隐藏层神经元数），即每一行为该样本某一时刻的隐藏层的状态值。

第 339、340 行：生成 RBM 对象。

- ☐ `input`: 设计矩阵。
- ☐ `n_visible`: 可见层数量，由于我们使用 MNIST 数据集，所以为 784 (28×28)。
- ☐ `n_hidden`: 隐藏层神经元数量。
- ☐ `numpy_rng`: numpy 对应的随机数生成引擎。
- ☐ `theano_rng`: theano 对应的随机数生成引擎。

第 343 行：获取该 RBM 的代价函数计算函数和参数更新规则。

第 349~351 行：如果没有图像输出目录，则创建该目录。

第 355 行：定义 Theano 函数 `train_rbm`，用于训练受限玻尔兹曼机，其参数为：样本索引、代价函数计算函数、参数更新规则，输入信号为一个迷你批次中的所有样本。

第 365 行：由于我们要记录训练所用时间，而且在训练过程中还要进行图像绘制，因此需要减去图像绘制时间，这里用 `plotting_time` 来记录图像绘制时间。

第 366 行：记录训练开始时间。

第 369 行：循环至最大训练样本遍历次数。

第 372 行：定义保存每个迷你批次平均代价函数值的列表 `mean_cost`。

第 373、374 行：循环处理每个迷你批次，调用 Theano 函数 `train_rbm`，求出代价函数值和参数更新值。

第 376 行：打印当前是第几次训练样本集遍历，以及对应的代价函数值是多少。

第 379 行：记录图像绘制（将图像内容保存到文件）的开始时间。

第 381~388 行：定义一个 10×10 平铺的图像网格，每个图像大小为 28×28 ，值来自 RBM 的连接权值矩阵。在本例中，连接权值矩阵为 784×500 ，在第 383 行将其转置后，变为 500×784 ，这时每一行恰好为一幅 28×28 的图像。我们采用 10×10 的网格图像，表示第 1 行第 1 列的图像为转置后权值矩阵的第 1 行，第 1 行第 2 列的图像为转置后权值矩阵的第 2 行，以此类推，最后一个图像为转置后权值矩阵的第 100 行。当然，这里其实并没有显示所有权值矩阵内容，只是显示了前 100 行，而不是全部 500 行。

第 389 行：将图像数据保存到 `filters_at_epoch_*.png` 图像文件中。

第 390 行：记录图像绘制结束时间。

第 391 行：将本次图像绘制时间累加到总的图像绘制时间中。

第 393 行：当训练结束后，记录训练结束时间。

第 395 行：求出训练时间，为训练结束时间减去训练开始时间再减去总的图像绘制时间，即 `pretraining_time`。在下一章中，我们将用受限玻尔兹曼机堆叠来搭建深度信念网络，在这个架构中，是先训练每个 RBM 网络，然后将这些 RBM 网络堆叠起来，形成深度信念网络，最后通过类似多层感知器算法进行调优。在这种体系下，RBM 网络的训练过程称为预训练，所以这里会采用这个名称。

由于受限玻尔兹曼机是生成式网络，其可以通过采样方法得到样本数据，所以下面的程序是演示 RBM 的采样过程。

第 402 行：求出测试样本集样本数。

第 405 行：随机选取一个测试样本集索引号 `test_idx`。

第 406~411 行：由于我们定义并行的吉布斯马可夫链数量为 `n_chains`，所以从 `test_idx` 中取出 `n_chains` 个测试样本。

第 412 行：每 `plot_every` 步采样绘制一次。

第 416~431 行：采用 Theano 中的 `scan` 函数进行循环，每次循环运行吉布斯采样可见层到隐藏层再到可见层，即 `RBM.gibbs_vhv` 函数，将最后一步状态赋给 `persistent_vis_chain`，

运行 `plot_every` 步。该循环将返回一个列表，每项中包括隐藏层线性和、隐藏层输出值、隐藏层采样值、可见层线性和、可见层输出值、可见层采样值和参数更新规则。

第 435 行：将最后一步可见层采样值赋给 `persistent_vis_chain`。

第 439~447 行：定义 Theano 函数 `sample_fn`，其输入参数为空，输出值为可见层输出值和可见层采样值，每运行一次更新一下参数。

第 450~453 行：定义表示一个网格图像内容的数组 `image_data`。在本例中，其为 $29 \times 10 + 1$ 行 $29 \times n_chains - 1$ 列，存储图像中的像素点的值，即形成 `n_samples` 行 `n_chains` 列的 28×28 的图像，程序中用 29 是因为需要留出 1 像素的间隔。

第 454 行：对行循环 `n_samples` 次。

第 457 行：对可见层进行采样，得到可见层输出值和可见层采样值。

第 459~464 行：生成 1 行 `n_chains` 列 28×28 的网格图像，图像中像素点的值为可见层神经元的输出值。

第 466~468 行：生成图像并将其保存到文件中。

程序的运行入口如下：

```
470 if __name__ == '__main__':
471     test_rbm()
```

运行结果如下：

```
... loading data
Training epoch 0, cost is -90.8953249282
Training epoch 1, cost is -80.2811759062
Training epoch 2, cost is -74.5895232892
Training epoch 3, cost is -72.4198624452
Training epoch 4, cost is -68.2973375487
Training epoch 5, cost is -63.8231670199
Training epoch 6, cost is -64.5855926857
Training epoch 7, cost is -67.5093626697
Training epoch 8, cost is -68.450707592
Training epoch 9, cost is -65.031722837
Training epoch 10, cost is -61.4680010826
Training epoch 11, cost is -61.603971751
Training epoch 12, cost is -64.0134390295
Training epoch 13, cost is -63.2977506849
Training epoch 14, cost is -62.642156477
Training took 138.988547 minutes
... plotting sample 0
... plotting sample 1
... plotting sample 2
... plotting sample 3
... plotting sample 4
... plotting sample 5
... plotting sample 6
... plotting sample 7
... plotting sample 8
... plotting sample 9
```

图 10.6 是一个 10×10 平铺的图像网格，每个图像大小为 28×28 ，图像中像素点的值来自 RBM 的连接权值矩阵。在本例中，连接权值矩阵为 784×500 ，经过转置后变为 500×784 ，这时每一行恰好为一幅 28×28 的图像。我们采用 10×10 的网格图像，表示第 1 行第 1 列的图像为转置后权值矩阵的第 1 行，第 1 行第 2 列的图像为转置后权值矩阵的第 2 行，以此类推，最后一个图像为转置后权值矩阵的第 100 行。当然，这里我们其实并没有显示所有权值矩阵内容，只是显示了前 100 行，而不是全部的 500 行。

由于刚开始训练，所以这些权值矩阵是相对随机的，看不出任何规律。

当训练样本集遍历 15 次之后，可以得到如图 10.7 所示的图像。

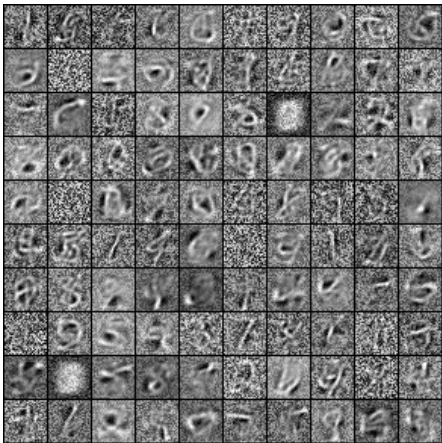


图 10.6 训练开始时的连接权值

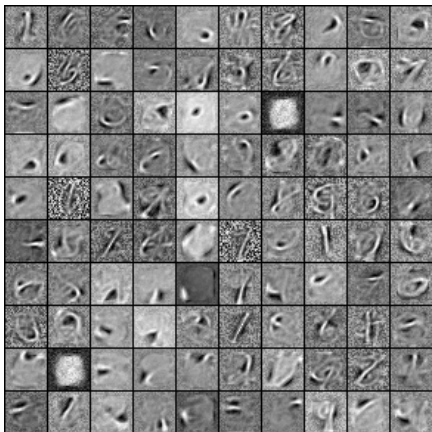


图 10.7 训练结束时的权值矩阵

当训练结束时，我们再看权值矩阵就隐约可以看到一些有趣的模式了，比如第 1 行第 1 列就是一个“1”字，最后一行倒数第 3 个就是一个“9”字。仔细观察，会发现很多图像都是一个数字，这就说明我们的 RBM 网络已经学习了 MNIST 数据集的相关知识了。

受限玻尔兹曼机是一种生成式网络，生成式学习的一个特点是，可以通过对其进行采样，生成与实际情况非常相似的样本。图 10.8 就是本例中生成的样本。



图 10.8 可见层采样生成的样本

由上图可以看出，通过对可见层采样，生成了保真度极高的样本，这些样本和 MNIST 数据集中的样本非常相似，我们几乎看不出什么差别。这就是生成学习的优点，在缺乏训练样本的情况下，其可以自动生成一些训练样本。只通过受限玻尔兹曼机的介绍，大家可能对生成式网络还没有什么概念，在机器学习基础部分，我们将专门用一章来讲一下生成式学习算法，包括高斯分布分析和贝叶斯网络，到时大家就会对生成式网络有一个系统的了解了。

上面的程序中用到了 MNIST 手写数字识别数据集载入，代码如下：

```
1 from __future__ import print_function
2 __docformat__ = 'restructuredtext en'
3 import six.moves.cPickle as pickle
4 import gzip
5 import os
6 import sys
7 import timeit
8 import numpy
9 import theano
10 import theano.tensor as T
11
12 class MnistLoader(object):
13     def load_data(self, dataset):
14         data_dir, data_file = os.path.split(dataset)
15         if data_dir == "" and not os.path.isfile(dataset):
16             new_path = os.path.join(
17                 os.path.split(__file__)[0],
18                 "..",
19                 "data",
20                 dataset
21             )
22             if os.path.isfile(new_path) or data_file == 'mnist.pkl.gz':
23                 dataset = new_path
24
25         if (not os.path.isfile(dataset)) and data_file == 'mnist.pkl.gz':
26             from six.moves import urllib
27             origin = (
28                 'http://www.iro.umontreal.ca/~lisa/deep/data/'
29                 'mnist/mnist.pkl.gz'
30             )
31             print('Downloading data from %s' % origin)
32             urllib.request.urlretrieve(origin, dataset)
33
34         print('... loading data')
35         # Load the dataset
36         with gzip.open(dataset, 'rb') as f:
37             try:
38                 train_set, valid_set, test_set = pickle.load(f,
39                                                             encoding='latin1')
40             except:
41                 train_set, valid_set, test_set = pickle.load(f)
42         def shared_dataset(data_xy, borrow=True):
43             data_x, data_y = data_xy
44             shared_x = theano.shared(numpy.asarray(data_x,
45                                                     dtype=theano.config.floatX),
46                                     borrow=borrow)
47             shared_y = theano.shared(numpy.asarray(data_y,
48                                                     dtype=theano.config.floatX),
49                                     borrow=borrow)
50             return shared_x, T.cast(shared_y, 'int32')
51
52         test_set_x, test_set_y = shared_dataset(test_set)
53         valid_set_x, valid_set_y = shared_dataset(valid_set)
54         train_set_x, train_set_y = shared_dataset(train_set)
55
56         rval = [(train_set_x, train_set_y), (valid_set_x, valid_set_y),
57                 (test_set_x, test_set_y)]
58         return rval
```

第 13 行：定义数据载入接口，参数为 MNIST 手写数字识别数据集文件。

第 14~33 行：MNIST 手写数字识别数据集文件如果存在则打开该文件，如果不存在则从指定网址下载该文件。

第 52~54 行：将训练样本集、验证样本集、测试样本集定义为 Theano 的共享变量，以便在 Theano 预编译函数中使用。

第 56~58 行：以指定格式返回 MNIST 手写数字识别数据内容。

在上面的程序中，还使用了 tile_raster_images 生成图像，其实现代码如下：

```

1 import numpy
2
3 def scale_to_unit_interval(ndar, eps=1e-8):
4     """ 将数组中所有数据进行缩放处理，变为[0,1]之间的数 """
5     ndar = ndar.copy()
6     ndar -= ndar.min()
7     ndar *= 1.0 / (ndar.max() + eps)
8     return ndar
9
10 def tile_raster_images(X, img_shape, tile_shape, tile_spacing=(0, 0),
11                        scale_rows_to_unit_interval=True,
12                        output_pixel_vals=True):
13     """
14     将每行代表一个图像的数组转变为平铺的网格图像。对于针对图像处理的神经
15     网络，尤其是第一层到第二层的连接权值矩阵，第一层为图像分辨率，例如
16     MNIST 手写数字识别数据集中，第一层将为 28×28=784 个神经元，第二层假设
17     有 500 个神经元，则其连接权值矩阵为 784 行 500 列，将其进行转置，将
18     变为 500 行 784 列，这时每一行代表 28×28 的图像，这样就可以利用这个函数
19     来进行处理
20     参数
21     - X: 2 维数组，每行代表一个样本；或者是一个具有四个元素的元组，每个元组
22       元素为 2 维数组
23     - img_shape: 图像的尺寸，如 MNIST 手写数字识别中就是 28×28
24     - tile_shape: 为几行几列平铺的图像
25     - tile_spacing: 图像之间的间隔
26     - scale_row_to_unit_interval: 是否规整化为 [0,1]
27     - output_pixel_values:
28     返回值
29     - 表示图像的 2 维数组
30     """
31     assert len(img_shape) == 2
32     assert len(tile_shape) == 2
33     assert len(tile_spacing) == 2
34
35     # The expression below can be re-written in a more C style as
36     # follows :
37     #
38     # out_shape = [0,0]
39     # out_shape[0] = (img_shape[0]+tile_spacing[0])*tile_shape[0] -
40     #                 tile_spacing[0]
41     # out_shape[1] = (img_shape[1]+tile_spacing[1])*tile_shape[1] -
42     #                 tile_spacing[1]
43     out_shape = [
44         (ishp + tsp) * tshp - tsp
45         for ishp, tshp, tsp in zip(img_shape, tile_shape, tile_spacing)
46     ]
47
48     if isinstance(X, tuple):
49         assert len(X) == 4
50         # Create an output numpy ndarray to store the image
51         if output_pixel_vals:
52             out_array = numpy.zeros((out_shape[0], out_shape[1], 4),
53                                    dtype='uint8')
54         else:
55             out_array = numpy.zeros((out_shape[0], out_shape[1], 4),
56                                    dtype=X.dtype)
57
58         # colors default to 0, alpha defaults to 1 (opaque)
59         if output_pixel_vals:
60             channel_defaults = [0, 0, 0, 255]
61         else:
62             channel_defaults = [0., 0., 0., 1.]
63

```

```

64     for i in range(4):
65         if X[i] is None:
66             # if channel is None, fill it with zeros of the correct
67             # dtype
68             dt = out_array.dtype
69             if output_pixel_vals:
70                 dt = 'uint8'
71             out_array[:, :, i] = numpy.zeros(
72                 out_shape,
73                 dtype=dt
74             ) + channel_defaults[i]
75         else:
76             # use a recurrent call to compute the channel and store it
77             # in the output
78             out_array[:, :, i] = tile_raster_images(
79                 X[i], img_shape, tile_shape, tile_spacing,
80                 scale_rows_to_unit_interval, output_pixel_vals)
81     return out_array
82
83     else:
84         # if we are dealing with only one channel
85         H, W = img_shape
86         Hs, Ws = tile_spacing
87
88         # generate a matrix to store the output
89         dt = X.dtype
90         if output_pixel_vals:
91             dt = 'uint8'
92         out_array = numpy.zeros(out_shape, dtype=dt)
93
94         for tile_row in range(tile_shape[0]):
95             for tile_col in range(tile_shape[1]):
96                 if tile_row * tile_shape[1] + tile_col < X.shape[0]:
97                     this_x = X[tile_row * tile_shape[1] + tile_col]
98                     if scale_rows_to_unit_interval:
99                         # if we should scale values to be between 0 and 1
100                         # do this by calling the 'scale_to_unit_interval'
101                         # function
102                         this_img = scale_to_unit_interval(
103                             this_x.reshape(img_shape))
104                     else:
105                         this_img = this_x.reshape(img_shape)
106                     # add the slice to the corresponding position in the
107                     # output array
108                     c = 1
109                     if output_pixel_vals:
110                         c = 255
111                     out_array[
112                         tile_row * (H + Hs): tile_row * (H + Hs) + H,
113                         tile_col * (W + Ws): tile_col * (W + Ws) + W
114                     ] = this_img * c
115     return out_array

```

第 43~46 行：生成输出图像的尺寸。我们以 MNIST 手写数字识别为例，原始图像尺寸为 28×28，平铺图像间隔为 1 像素，要生成 10×10 平铺图像，则输出图像行数=(原始图像行数+行方向间隔)×平铺图像行数-行方向间隔=(28+1)×28-1，减 1 是因为最后一行平铺图像就不需要底下的行方向间隔了。同理，可以计算输出图像列数=(原始图像列数+列方向间隔)×平铺图像列数-1=(28+1)×10-1。

第 48 行：判断原始图像是否为彩色图像，如果是彩色图像则为四元组，分别为 R、G、B、A 的值。在这个例子中是黑白图像，所以将执行第 83 行的操作。

第 85 行：求出图像的高度和宽度。

第 86 行：求出平铺图像在高度和宽度方向的间隔。

第 89 行：定义输出数组的类型为原始信号类型。

第 90、91 行：如果输出像素值参数为 True，则返回值数组类型为非符号字节型。

第 92 行：生成二维数组作为返回值。

第 94、95 行：对返回值数组对应平铺图像位置的每行每列进行循环。

第 96 行：输入参数 `X`，每行代表一个图像，平铺图像第 `tile_row` 行第 `tile_col` 列，则代表 `X` 中的行数=`tile_row`×平铺图像列数+`tile_col`，所以该值必须小于 `X` 中的行数，这样才会执行下面的操作。

第 97 行：将 `X` 中第 `tile_row`×平铺图像列数+`tile_col` 行的数据取出赋给 `this_x`，即将一幅图像的数据赋给 `this_x`。

第 98 行：根据参数 `scale_rows_to_unit_interval` 判断是否需要规整化。

第 102、103 行：如果需规整到[0,1]，则先将一维数组 `this_x` 变为二维数组，形状为参数 `img_shape` 规定的形状。然后调用 `scale_to_unit_interval` 函数进行规整化，将这个二维数组赋给 `this_img`。

第 105 行：如果不需要规整化，定义 `this_img` 为将一维数组 `this_x` 变为二维数组，形状为参数 `img_shape` 规定形状后的二维数组。

第 109、110 行：根据参数判断是否需要输出像素值，如果需要则 `c=255`，即输出的值为 0~255 间的值。

第 111~114 行：将求出的 `this_img` 逐像素赋给返回值数组对应的图像位置的像素值。

第 115 行：返回该二维数组。

第 11 章

深度信念网络

在上一章中，我们介绍了受限玻尔兹曼机，并说明它是一种无监督、生成式网络。但是受限玻尔兹曼机很少单独使用，它总是作为某些深度网络的一层。本章将介绍深度信念网络，就是将预训练的受限玻尔兹曼机堆叠起来，通过类似多层感知器的学习算法进行微调，达到非常好的效果。这种网络是 2006 年由 Hinton 提出的，它开启了深度学习复兴的先河。虽然当前已经较少采用这种网络，但它还是一种非常重要的网络，而且从 2016 年的情况来看，深度信念网络的应用有重新崛起之势。

在这一章中，我们将介绍深度信念网络的 Theano 实现，同时也会介绍其在 MNIST 手写数字识别中的应用。

11.1 深度信念网络原理

预训练好的受限玻尔兹曼机可以进行堆叠，形成深度信念网络。深度信念网络是一种图模型，可以通过深度层次结构，来发现训练数据的高效表示方式。其可以表示输入信号向量 \mathbf{x} 和共有 L 个隐藏层前向神经网络。假设第 ℓ 个隐藏层状态为 $\mathbf{h}^{(\ell)}$ ，有以下公式：

$$P(\mathbf{x}, \mathbf{a}^{(1)}, \mathbf{a}^{(2)}, \dots, \mathbf{a}^{(L)}) = \left(\prod_{\ell=0}^{L-2} P(\mathbf{a}^{(\ell)} | \mathbf{a}^{(\ell+1)}) \right) P(\mathbf{a}^{(L-1)}, \mathbf{a}^{(L)})$$

式中， $\mathbf{a}^{(0)} = \mathbf{x}$ ， $P(\mathbf{a}^{(\ell)} | \mathbf{a}^{(\ell+1)})$ 表示第 ℓ 层以隐藏层为条件可见层的概率密度。 $P(\mathbf{a}^{(L-1)}, \mathbf{a}^{(L)})$ 则表示最上层 RBM 的可见层和隐藏层的概率密度。

深度信念网络的架构如图 11.1 所示。

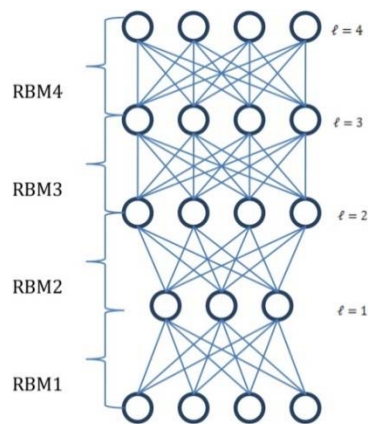


图 11.1 深度信念网络架构

由图 11.1 可以看出，其实际上是由 4 个受限玻尔兹曼机构成的，下一层 RBM 的隐藏层作为上一层 RBM 的可见层，这 4 个 RBM 可以先从下向上分别训练，再堆叠在一起，最后就形成了深度信念网络，具体学习过程如下。

- (1) 训练第一个受限玻尔兹曼机网络，将输入信号加载到可见层，即 $\mathbf{a}^{(0)} = \mathbf{x}$ 。
- (2) 训练完成后，每个训练样本在第一个受限玻尔兹曼机隐藏层的状态可以视为该样本的一个特征表示，可以通过平均激活 $P(\mathbf{a}^{(1)} = 1 | \mathbf{a}^{(0)})$ ，或者通过对隐藏层采样 $P(\mathbf{a}^{(1)} | \mathbf{a}^{(0)})$ 来得到。
- (3) 训练第二个 RBM，将第一个 RBM 的隐藏层状态作为可见层输入，对其进行训练。
- (4) 重复上述操作，将所有 RBM 训练完成，这就完成了预训练过程。
- (5) 当预训练完所有 RBM 后，就进入网络微调阶段。可以将最后一个 RBM 的隐藏层状态接入到其他监督学习网络中，通过监督学习对网络进行微调。实际中比较常用的方法就是先接入上一个逻辑回归层，然后将其视为一个普通的多层感知器，利用 BP 算法进行学习，微调连接权值。

11.2 深度信念网络 TensorFlow 实现

深度信念网络实际由一系列逐层预训练好的受限玻尔兹曼机堆叠而成，形成典型的多层感知器模型，并利用多层感知器模型的算法进行模型调优。理解了受限玻尔兹曼机和多层感知器模型之后，理解深度信念网络就比较简单了。

我们首先来看 MNIST 手写数字识别数据集载入，代码如下：

```
1 def load_datasets(self):
2     mnist = input_data.read_data_sets(self.datasets_dir,
3     one_hot=True)
4     X_train = mnist.train.images
5     y_train = mnist.train.labels
6     X_validation = mnist.validation.images
```

```

7      y_validation = mnist.validation.labels
8      X_test = mnist.test.images
9      y_test = mnist.test.labels
10     return X_train, y_train, X_validation, y_validation, \
11           X_test, y_test, mnist

```

第 2、3 行：调用 TensorFlow 的 `input_data` 的 `read_data_sets` 方法，第一个参数为数据集存放路径，第二个参数是标签集的格式。在原始 MNIST 数据集中，我们知道每个样本是 28×28 的黑白图片，对应的是 0~9 的数字标签，所以其格式为：[...784 (28×28) 像素点的值...][3]。其中，第一项为 784 (28×28) 个 0~1 的浮点数，0 代表黑色，1 代表白色；第二项的“3”代表这个样本是数字 3。为了后续处理方便，我们将标签[3]改为 one-hot 形式，因为标签代表 0~9 的数字，所以标签集为 10 维向量，每维上取值为 0 代表不是这个对应位置的数字，取值为 1 代表是这个对应位置的数字。其中，只有一维可以取 1，因此称之为 one-hot，还以上面的例子为例，标签集的格式就变为：[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]，因为第四位为 1，所以代表这个样本是数字 3。

第 4 行：取出训练样本集输入信号集 X_{train} ，其为设计矩阵形式，每一行代表一个样本，行数为训练样本集中的样本数量。在这个例子中，就 $X_{\text{train}} \in \mathbb{R}^{55000 \times 784}$ ，其中训练样本集中有 55000 个样本，每个样本是 784 (28×28) 维的图片。

第 5 行：取出训练样本集标签集 y_{train} ，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为训练样本集中的样本数量。在这个例子中，就 $y_{\text{train}} \in \mathbb{R}^{55000 \times 10}$ ，其中训练样本集中有 55000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 6 行：取出验证样本集输入信号集 $X_{\text{validation}}$ ，其为设计矩阵形式，每一行代表一个样本，行数为验证样本集中的样本数量。在这个例子中，就 $X_{\text{validation}} \in \mathbb{R}^{5000 \times 784}$ ，其中验证样本集中有 5000 个样本，每个样本是 784 (28×28) 维的图片。根据前面我们的讨论可以知道，在训练过程中，为了防止模型出现过拟合，模型的泛化能力降低（模型在训练样本集达到非常高的精度，但是在未见过的测试样本集或实际应用中，精度反而不高），通常会采用 Early Stopping 策略，就是在逻辑回归模型训练过程中，只用训练样本集对模型进行训练，每隔一定的时间间隔，计算模型在未见过的验证样本集上识别的精度，并记录迄今为止在验证样本集上取得的最高精度。我们会发现，在训练初期，验证样本集上的识别精度会稳步提高，但是到了一定阶段之后，验证样本集上的识别精度就不会再明显提高了，甚至开始逐渐下降，这就说明模型出现了过拟合，这时就可以停止模型训练，将在验证样本集上取得最佳识别精度的模型参数作为模型最终的参数。综上所述，验证样本集主要用于防止模型出现过拟合，为 Early Stopping 算法提供终止依据。

第 7 行：取出验证样本集标签集 $y_{\text{validation}}$ ，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为训练样本集中的样本数量。在这个例子中，就 $y_{\text{validation}} \in \mathbb{R}^{5000 \times 10}$ ，其中验证样本集中有 5000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 8 行：取出测试样本集输入信号集 X_{test} ，其为设计矩阵形式，每一行代表一个样本，行数为训练样本集中的样本数量。在这个例子中，就 $X_{\text{test}} \in \mathbb{R}^{10000 \times 784}$ ，其中训练样本集

中有 10000 个样本，每个样本是 784（28×28）维的图片。测试样本集主要用于模型训练结束后对模型性能进行评估。由于模型没有见过测试样本集中的样本，可以模拟模型在实际部署之后的情况，模型在测试样本集上的识别精度，基本可以视为模型在实际应用中可以达到的精度。

第 9 行：取出测试样本集标签集 y_test ，其为 one-hot 为行的矩阵，每一行代表对应样本的正确结果，行数为测试样本集中的样本数量。在这个例子中，就 $y_test \in \mathbb{R}^{10000 \times 10}$ ，其中测试样本集中有 10000 个样本，每个正确结果为 10 维的 one-hot 向量，代表 0~9 的数字。

第 10、11 行：返回训练样本集输入信号集、训练样本集标签集、验证样本集输入信号集、验证样本集标签集、测试样本集输入信号集、测试样本集标签集。

下面来看深度信念网络模型的训练方法，代码如下：

```
1 def train(self, mode=TRAIN_MODE_NEW, ckpt_file='work/dbn.ckpt'):
2     X_train, y_train, X_validation, y_validation, \
3         X_test, y_test, mnist = self.load_datasets()
4     self.pretrain(X_train, X_validation)
5     if self.mlp_engine is None:
6         self.mlp_engine = Mlp_Engine(self.rbms, 'datasets')
7     self.mlp_engine.train()
```

第 2、3 行：读入 MNIST 手写数字识别数据集，包括：训练样本集输入样本集、训练样本集标签集、验证样本集输入样本集、验证样本集标签集、测试样本集输入样本集、测试样本集标签集。

第 4 行：调用预训练方法。深度信念网络训练包括两个阶段：第一阶段是逐层训练受限玻尔兹曼机，第二阶段是将预训练好的受限玻尔兹曼机堆叠在一起，形成一个标准的多层感知器模型进行调优。本行就是逐层预训练受限玻尔兹曼机。

第 5、6 行：如果没有初始化多层感知器模型，则初始化多层感知器模型。

第 7 行：调用多层感知器模型的训练方法，对参数进行调优。

接下来看网络预训练方法，在这里需要逐层训练受限玻尔兹曼机。为了更好地讲解这一过程，先来看一下模型的构造函数，代码如下：

```
1 def __init__(self):
2     self.datasets_dir = 'datasets/'
3     self.random_seed = 1
4     self.rbm_W = []
5     self.rbm_b = []
6     self.rbms = []
7     self.rbm_graphs = []
8     self.layers = [1024, 784, 512, 256]
9     self.name = 'dbn'
10    prev = 784
11    self.mlp_engine = None
12    for idx, layer in enumerate(self.layers):
13        rbm_str = 'rbm_' + str(idx+1)
14        name = self.name + '_' + rbm_str
15        tf_graph = tf.Graph()
```

```

16     self.rbms.append(Rbm_Engine(name, tf_graph=tf_graph, n=prev,
17                               num_hidden=layer))
18     prev = layer
19     self.rbm_graphs.append(tf_graph)

```

第 2 行：指定数据集存放位置。

第 3 行：随机数生成的种子，在网络权值和偏置值初始化时都会用随机数进行初始化。如果每次生成的随机数均不相同，那么调试起来会非常不方便。因此这里采用随机数种子，这样生成的随机数就是一样的，便于进行调试。

第 4 行：连接权值矩阵列表，每个预训练的受限玻尔兹曼机对应一个元素。

第 5 行：偏置值列表，每个预训练的受限玻尔兹曼机对应一个元素。

第 6 行：逐层预训练的受限玻尔兹曼机列表。

第 7 行：逐层预训练的受限玻尔兹曼机所对应的 Graph 列表，每个受限玻尔兹曼机对应一个 Graph 对象。

第 8 行：定义深度信念网络架构，第一个受限玻尔兹曼机结构为 784—1024—784；第二个受限玻尔兹曼机结构为 1024—784—1024；第三个受限玻尔兹曼机结构为 784—512—784；第四个受限玻尔兹曼机结构为 512—256—512。

第 9 行：定义模型名称。

第 10 行：定义输入层维度为 784。

第 11 行：定义用于调优阶段的多层感知器模型。

第 12 行：利用第 13~19 行循环初始化深度信念网络中的各个受限玻尔兹曼机。

第 13、14 行：定义受限玻尔兹曼机名称，因为需要多个受限玻尔兹曼机的堆叠，这样可以区分不同的受限玻尔兹曼机。

第 15 行：建立一个新的 TensorFlow 计算图。

第 16、17 行：生成一个新的受限玻尔兹曼机，并添加到受限玻尔兹曼机列表中。

第 18 行：将本受限玻尔兹曼机隐藏层作为下一个受限玻尔兹曼机的可见层。

第 19 行：将生成的 TensorFlow 的 Graph 添加到 Graph 列表中。

下面来看预训练过程，代码如下：

```

1 def pretrain(self, X_train, X_validation):
2     X_train_prev = X_train
3     X_validation_prev = X_validation
4     for idx, rbm in enumerate(self.rbms):
5         print('pretrain:{0}'.format(rbm.name))
6         tf_graph = self.rbm_graphs[idx]
7         X_train_prev, X_validation_prev = self.pretrain_rbm(
8             self.rbm_graphs[idx], rbm,
9             X_train_prev, X_validation_prev)
10    return X_train_prev, X_validation_prev

```

第 2、3 行：每个受限玻尔兹曼机的输入均为上一个受限玻尔兹曼机隐藏层的内容，初始时将上一个受限玻尔兹曼机隐藏层的值设置为训练样本集和验证样本集。

第 4 行：使所有受限玻尔兹曼机列表中的受限玻尔兹曼机循环第 5~9 行的操作。

第 6 行：取出该受限玻尔兹曼机对应的 TensorFlow 的 Graph。

第 7~9 行：预训练某个单独的受限玻尔兹曼机。

第 10 行：返回整个受限玻尔兹曼机组的隐藏层状态。

下面来看每个受限玻尔兹曼机的训练过程，代码如下：

```
1 def pretrain_rbm(self, graph, rbm, X_train, X_validation):
2     rbm.train(X_train, X_validation)
3     X_train_next = rbm.transform(graph, X_train)
4     X_validation_next = rbm.transform(graph, X_validation)
5     return X_train_next, X_validation_next
```

第 2 行：调用受限玻尔兹曼机的训练方法，进行预训练。

第 3 行：调用受限玻尔兹曼机的 `transform` 方法，将训练样本集转变为其隐藏层输出的结果，用于对下一个受限玻尔兹曼机的输入。

第 4 行：调用受限玻尔兹曼机的 `transform` 方法，将验证样本集转变为其隐藏层输出的结果，用于对下一个受限玻尔兹曼机的输入。

第 5 行：返回隐藏层的输出，作为下一个受限玻尔兹曼机的输入。

以上就是深度信念网络预训练的全部过程，这里用到了与上一章有一定差别的受限玻尔兹曼机模型。为了使读者有一个完整的认识，我们重新讲解一下这个受限玻尔兹曼机模型。

首先来看构造函数，代码如下：

```
1 def __init__(self, name='rbm', tf_graph=None, n=784, num_hidden=250):
2     self.datasets_dir = 'datasets/'
3     self.random_seed = 1 # 用于测试，使每次生成的随机数相同
4     self.name = name
5     #self.loss_func = loss_func
6     self.learning_rate = 0.0001
7     self.num_epochs = 10
8     self.batch_size = 128
9     self.regtype = 'l2'
10    self.regcoef = 0.00001
11    #self.loss = Loss(self.loss_func)
12    self.num_hidden = num_hidden
13    self.visible_unit_type = 'bin'
14    self.gibbs_sampling_steps = 3
15    self.stddev = 0.1
16    self.W = None
17    self.bh_ = None
18    self.bv_ = None
19    self.w_upd8 = None
20    self.bh_upd8 = None
21    self.bv_upd8 = None
22    self.cost = None
23    self.input_data = None
24    self.hrand = None
25    self.vrand = None
26    self.tf_graph = tf_graph
27    self.n = n # 28*28
```

第 2 行：定义数据集文件存放路径。

第 3 行：定义随机数生成种子，以种子方式生成的随机数每次生成的一样，这样便于

对神经网络的调试。

第 4 行：定义网络的名称，因为在深度信念网络中，我们会将预训练好的受限玻尔兹曼机堆叠起来，为了区分不同的受限玻尔兹曼机，需要给其起不同的名字。

第 6 行：定义学习率。

第 7 行：定义完整学习整个训练样本集的遍数。

第 8 行：迷你批次中的样本数量。

第 9 行：调整项类型，在这里使用 L2 调整项，即权值衰减项。

第 10 行：定义 L2 调整项（权值衰减）的系数。

第 12 行：定义隐藏层神经元数量。

第 13 行：定义可见层神经元类型，经典受限玻尔兹曼机的可见层是二进制类型的，如本例所示。而现在经过扩展的受限玻尔兹曼机可以支持连续型变量。

第 14 行：吉布斯采样次数。

第 15 行：定义正态分布中的标准差。

第 16 行：定义可见层和隐藏层之间的连接权值。

第 17 行：定义隐藏层的偏置值矩阵。

第 18 行：定义可见层的偏置值矩阵。

第 19 行：权值更新时 TensorFlow 计算图中的节点。

第 20 行：隐藏层偏置值更新时对应的 TensorFlow 计算图中的节点。

第 21 行：可见层偏置值更新时对应的 TensorFlow 计算图中的节点。

第 22 行：代价函数在 TensorFlow 计算图中的节点。

第 23 行：输入信号变量。

第 24 行：隐藏层对应的 placeholder。

第 25 行：可见层对应的 placeholder。

第 26 行：本模型对应的 TensorFlow 计算图。

第 27 行：输入向量维度。

下面来看受限玻尔兹曼机的模型创建过程，代码如下：

```
1 def build_model(self):
2     print('Build RBM Model')
3     self.X = tf.placeholder(shape=[None, self.n], dtype=tf.float32, name='X')
4     self.hrand = tf.placeholder(shape=[None, self.num_hidden],
5                                 dtype=tf.float32, name='h')
6     self.vrand = tf.placeholder(shape=[None, self.n],
7                                 dtype=tf.float32, name='v')
8     self.y = tf.placeholder(shape=[None, 10], dtype=tf.float32, name='y')
9     self.keep_prob = tf.placeholder(dtype=tf.float32, name='keep_prob')
10    #
11    self.W = tf.Variable(tf.truncated_normal(shape=[self.n, self.num_hidden],
12                                             mean=0.0, stddev=0.1), name='W')
13    self.bh_ = tf.Variable(tf.constant(0.1, shape=[self.num_hidden]),
14                           name='bh')
15    self.bv_ = tf.Variable(tf.constant(0.1, shape=[self.n], name='bv'))
16    #
17    self.encode, _ = self.sample_hidden_from_visible(self.X)
```



```

18 self.reconstruction = self.sample_visible_from_hidden(
19     self.encode, self.n)
20 hprob0, hstate0, vprob, hprob1, hstate1 = self.gibbs_sampling_step(
21     self.X, self.n)
22 self.vprob = vprob
23 self.hprob = hprob1
24 positive = self.compute_positive_association(self.X,
25     hprob0, hstate0)
26 nn_input = vprob
27 for step in range(self.gibbs_sampling_steps - 1):
28     hprob, hstate, vprob, hprob1, hstate1 = self.gibbs_sampling_step(
29         nn_input, self.n)
30     nn_input = vprob
31 negative = tf.matmul(tf.transpose(vprob), hprob1)
32 #
33 self.w_upd8 = self.W.assign_add(
34     self.learning_rate * (positive - negative) / self.batch_size)
35 self.bh_upd8 = self.bh_.assign_add(tf.multiply(self.learning_rate,
36     tf.reduce_mean(tf.subtract(hprob0, hprob1), 0)))
37 self.bv_upd8 = self.bv_.assign_add(tf.multiply(self.learning_rate,
38     tf.reduce_mean(tf.subtract(self.X, vprob), 0)))
39 clip_inf = tf.clip_by_value(vprob, 1e-10, float('inf'))
40 clip_sup = tf.clip_by_value(1 - vprob, 1e-10, float('inf'))
41 loss = - tf.reduce_mean(tf.add(
42     tf.multiply(self.X, tf.log(clip_inf)),
43     tf.multiply(tf.subtract(1.0, self.X),
44         tf.log(clip_sup))))
45 self.cost = loss + self.regcoef*(tf.nn.l2_loss(self.W) +
46     tf.nn.l2_loss(self.bh_) + tf.nn.l2_loss(self.bv_))

```

第 3 行：定义用于存放输入信号的 `placeholder`，其为包含一个迷你批次的设计矩阵，第一维是迷你样本集中的序号，第二维为样本特征值向量。

第 4、5 行：定义初始化隐藏层为随机数时用的 `placeholder`，第一维为迷你批次中的样本序号，第二维为隐藏层神经元数。

第 6、7 行：定义初始化可见层为随机数时用的 `placeholder`，第一维为迷你批次中的样本序号，第二维为可见层神经元数。

第 8 行：定义正确分类结果标签集的 `placeholder`，由于受限玻尔兹曼机为非监督学习，所以这个变量目前没有用到。

第 11、12 行：定义可见层与隐藏层之间的连接权值矩阵 `W`，用均值为 0.0、标准差为 0.1 的正态分布随机数进行初始化。

第 13、14 行：定义隐藏层神经元偏置值 `bh_`，用常数 0.1 进行初始化。

第 15 行：定义可见层神经元偏置值 `bv_`，用常数 0.1 进行初始化。

第 17 行：定义受限玻尔兹曼机编码过程，调用本类 `sample_hidden_from_visible` 方法，通过可见层状态求出隐藏层状态，可以视为对原始信号进行编码，或者提取出其中的特征。

第 18、19 行：定义重建算子，调用本类 `sample_visible_from_hidden` 方法，由隐藏层状态求出可见层状态，相当于通过特征恢复原始信号。

第 20、21 行：定义吉布斯采样算子，调用本类 `gibbs_sampling_step` 方法，即先由可见层求出隐藏层状态 0，再由隐藏层状态求出可见层状态，最后由新的可见层状态求出隐藏层

状态 1，返回值 `hprob0` 和 `hstate0` 分别代表第一次求出的隐藏层状态对应的概率值和由概率值转化而来的二进制状态，`vprob` 代表可见层概率值，`hprob1` 和 `hstate1` 分别代表由上一状态求出的隐藏层状态对应的概率值和由概率值转化而来的二进制状态。

第 22 行：将可见层概率值定义为 TensorFlow 算子。

第 23 行：将隐藏层的概率值定义为 TensorFlow 算子，并且将其传给下一个受限玻尔兹曼机的可见层。

第 24、25 行：计算正向状态，即第一次由可见层到隐藏层时，可见层状态与隐藏层状态的点积。

第 26 行：将当前可见层概率值作为输入参数。

第 27 行：循环第 28~30 行操作，执行 `self.gibbs_sampling_steps-1` 次吉布斯采样。

第 28、29 行：先由可见层求出隐藏层状态 0，再由隐藏层状态求出可见层状态，最后由新的可见层状态求出隐藏层状态 1，返回值 `hprob` 和 `hstate` 分别代表第一次求出的隐藏层状态对应的概率值和由概率值转化而来的二进制状态，`vprob` 代表可见层概率值，`hprob1` 和 `hstate1` 分别代表由上一状态求出的隐藏层状态对应的概率值和由概率值转化而来的二进制状态，`vprob` 代表可见层概率值。

第 30 行：将可见层概率值作为下一次吉布斯采样的输入。

第 31 行：将可见层概率值与最后的隐藏层概率值进行点积运算，作为负向阶段的状态。

第 33、34 行：定义 TensorFlow 算子更新可见层与隐藏层之间的连接权值。

第 35、36 行：定义 TensorFlow 算子更新隐藏层偏置值。

第 37、38 行：定义 TensorFlow 算子更新可见层偏置值。

第 39 行：为了提高数值计算精度和稳定性，求出可见层概率值规整化的值，使其在 $1e-10$ 到正无穷之间。

第 40 行：为了提高数值计算精度和稳定性，求出 $1-v$ 可见层概率值规整化的值，使其在 $1e-10$ 到正无穷之间。

第 41~44 行：定义代价函数为原始输入信号与经过吉布斯采样后可见层概率值之间的交叉熵，如下：

$$\text{loss} = x \log v_{\text{prob}} + (1 - x) \log(1 - v_{\text{prob}})$$

第 45、46 行：最终的代价函数为交叉熵加上 L2 调整项（权值衰减），这里同时调整可见层与隐藏层之间的连接权值、可见层偏置值和隐藏层偏置值。

下面来看模型的训练方法，代码如下：

```
1 def train(self, mode=TRAIN_MODE_NEW, ckpt_file='work/rbm.ckpt'):
2     X_train, y_train, X_validation, y_validation, X_test, \
3         y_test, mnist = self.load_datasets()
4     with self.tf_graph.as_default():
5         self.build_model()
6         saver = tf.train.Saver()
7         with tf.Session() as sess:
8             sess.run(tf.global_variables_initializer())
9             for epoch in range(self.num_epochs):
10                 np.random.shuffle(X_train)
```

```

11         batches = [_ for _ in self.gen_mini_batches(X_train,
12                                                         self.batch_size)]
13         total_batches = len(batches)
14         for idx, batch in enumerate(batches):
15             cost_val, _, _, _ = sess.run([self.cost, self.w_upd8,
16                                           self.bh_upd8, self.bv_upd8], feed_dict={
17                 self.X: batch,
18                 self.hrand: np.random.rand(batch.shape[0],
19                                             self.num_hidden),
20                 self.vrand: np.random.rand(batch.shape[0],
21                                             batch.shape[1])})
21             if (epoch*total_batches + idx) % 100 == 0:
22                 saver.save(sess, ckpt_file)
23                 print('{0}_{1}:cost={2}'.format(epoch, idx, cost_val))

```

第 2、3 行：载入 MNIST 数据集，得到训练样本集输入样本集、训练样本集标签集、验证样本集输入样本集、验证样本集标签集、测试样本集输入样本集、测试样本集标签集。

第 4 行：启动 TensorFlow 的 graph。

第 5 行：调用 build_model 方法创建受限玻尔兹曼机模型。

第 6 行：创建 TensorFlow 的 saver 对象，用于保存模型的参数和超参数。

第 7 行：启动 TensorFlow 会话。

第 8 行：初始化全局变量。

第 9 行：循环第 10~23 行操作，训练完整训练样本集指定遍数。

第 10 行：对训练样本集进行随机洗牌，打乱顺序。

第 11、12 行：以指定大小生成迷你批次列表 batches。

第 13 行：求出迷你批次总数。

第 14 行：循环第 15~23 行操作，处理每个迷你批次。

第 15~21 行：以本迷你批次样本集随机数初始化的隐藏层状态和随机数初始化的可见层状态为输入，求出模型的代价函数，网络会运行吉布斯采样并更新可见层和隐藏层之间的连接权值、可见层偏置值、隐藏层偏置值。

第 22、23 行：如果训练了 100 个迷你批次，则保存模型的参数和超参数到模型文件中。

第 24 行：打印当前的训练状态。

在训练方法中，我们用到了工具方法，下面分别进行介绍。

我们首先来看从可见层到隐藏层的采样方法，代码如下：

```

1 def sample_hidden_from_visible(self, vis_layer):
2     hprobs = tf.nn.sigmoid(tf.add(tf.matmul(vis_layer, self.W), self.bh_))
3     hstates = self.sample_prob(hprobs, self.hrand)
4     return hprobs, hstates
5
6 def sample_prob(self, probs, rand):
7     return tf.nn.relu(tf.sign(probs - rand))

```

第 2 行：由可见层状态求出隐藏层概率值，公式为：

$$h = \frac{1}{1 + e^{-(Wv+bh)}}$$

第 3 行：调用 `sample_prob` 方法，通过概率值求出二进制值表示的状态。

第 4 行：返回隐藏层的概率值和二进制值表示的状态。

第 6、7 行：定义 `sample_prob` 方法，将隐藏层节点的概率值与生成的 0~1 的随机值进行比较，如果隐藏层节点的概率值大就取 1，否则取 0。

下面来看由隐藏层采样可见层，代码如下：

```
1 def sample_visible_from_hidden(self, hidden, n_features):
2     visible_activation = tf.add(
3         tf.matmul(hidden, tf.transpose(self.W)),
4         self.bv_
5     )
6     if self.visible_unit_type == 'bin':
7         vprobs = tf.nn.sigmoid(visible_activation)
8     elif self.visible_unit_type == 'gauss':
9         vprobs = tf.truncated_normal(
10             (1, n_features), mean=visible_activation, stddev=self.stddev)
11     else:
12         vprobs = None
13     return vprobs
```

第 2~5 行：计算可见层的输入值。

第 6、7 行：如果可见层是二进制格式，采用 `Sigmoid` 函数计算概率值。

第 8、9 行：如果可见层为高斯型神经元，则采用以可见层输入值为均值、标准差为 0.1 的正态分布随机数作为可见层的概率值。

第 11、12 行：如果既不是二进制也不是高斯型，则概率值返回空。

第 13 行：返回可见层概率值。

下面来看吉布斯采样方法，代码如下：

```
1 def gibbs_sampling_step(self, visible, n_features):
2     hprobs, hstates = self.sample_hidden_from_visible(visible)
3     vprobs = self.sample_visible_from_hidden(hprobs, n_features)
4     hprobs1, hstates1 = self.sample_hidden_from_visible(vprobs)
5     return hprobs, hstates, vprobs, hprobs1, hstates1
```

第 2 行：由可见层采样隐藏层，得到隐藏层的概率值和状态。

第 3 行：由隐藏层采样可见层，得到可见层的概率值。

第 4 行：重新由可见层采样隐藏层，得到隐藏层新的概率值和状态。

第 5 行：返回第一次由可见层采样隐藏层时隐藏层的概率值和状态、可见层概率值，以及第二次由可见层采样隐藏层时隐藏层的概率值和状态。

下面来看求正向阶段可见层状态的方法，代码如下：

```
1 def compute_positive_association(self, visible,
2     hidden_probs, hidden_states):
3     if self.visible_unit_type == 'bin':
4         positive = tf.matmul(tf.transpose(visible), hidden_states)
5     elif self.visible_unit_type == 'gauss':
6         positive = tf.matmul(tf.transpose(visible), hidden_probs)
7     else:
8         positive = None
9     return positive
```

第 3、4 行：如果可见层为二进制类型，则使用隐藏层状态与可见层状态的乘积（形状为可见层维度×隐藏层维度）来表示。

第 5、6 行：如果可见层为高斯型，则使用隐藏层概率值与可见层状态的乘积（形状为可见层维度×隐藏层维度）来表示。

第 7、8 行：否则返回空。

第 9 行：返回计算出的值。

产生迷你批次的方法的代码如下：

```
1 def gen_mini_batches(self, X, batch_size):
2     X = np.array(X)
3     for i in range(0, X.shape[0], batch_size):
4         yield X[i:i + batch_size]
```

这段代码先将样本变为 `numpy` 的数组，将其分割为 `batch_size` 大小的子数组，用 `yield` 函数将其作为一个序列。

当预训练完所有的去噪自动编码之后，就进入了整体网络调优阶段，可以把这个阶段假想为在已经预训练好的受限玻尔兹曼机上叠加一个用于分类的 `Softmax` 回归层，形成一个完整的多层感知器模型，相当于已经知道了多层感知器模型的连接权值矩阵和偏置值，在此基础上继续训练多层感知器模型。

首先来看多层感知器模型的构造函数，代码如下：

```
1 def __init__(self, daes, datasets_dir):
2     self.datasets_dir = datasets_dir
3     self.batch_size = 100
4     self.n = 784
5     self.k = 10
6     self.L = np.array([self.n, 1024, 784, 512, 256, self.k])
7     self.lanmeda = 0.001
8     self.keep_prob_val = 0.75
9     self.daes = daes
10    self.model = {}
```

第 2 行：指定数据集文件存放目录。

第 3 行：指定迷你批次大小。

第 4 行：指定输入向量维度为 784。

第 5 行：输出信号类别为 10，即 0~9 这 10 个数字。

第 6 行：定义网络结构的第一层为 784 维，第二层为 1024 维，第三层为 784 维，第四层为 512 维，第五层为 256 维，第六层为 10 维，对应 0~9 这 10 个数字。

第 7 行：定义 L2 调整项（权值衰减）的系数。

第 8 行：定义采用 `Dropout` 调整技术时，神经元激活的概率，在本例中 75% 的神经元保持输出不变。

第 9 行：引入预训练好的去噪自动编码器，主要用于读取已经训练好的参数和超参数。

第 10 行：定义保存模型变量的属性。

接下来看模型构建过程，代码如下：

```
1 def build_model(self, mode='train'):
2     print('mode={0}'.format(mode))
```

```

3 self.X = tf.placeholder(tf.float32, [None, 784])
4 self.y = tf.placeholder(tf.float32, [None, 10])
5 self.keep_prob = tf.placeholder(tf.float32) #Dropout 失活率
6 if 'train' == mode:
7     # 取出预训练去噪自动编码器参数
8     with self.daes[0].tf_graph.as_default():
9         with tf.Session() as sess:
10             sess.run(tf.global_variables_initializer())
11             dae0_W1 = sess.run(self.daes[0].W1)
12             dae0_b2 = sess.run(self.daes[0].b2)
13         with self.daes[1].tf_graph.as_default():
14             with tf.Session() as sess:
15                 sess.run(tf.global_variables_initializer())
16                 dae1_W1 = sess.run(self.daes[1].W1)
17                 dae1_b2 = sess.run(self.daes[1].b2)
18         with self.daes[2].tf_graph.as_default():
19             with tf.Session() as sess:
20                 sess.run(tf.global_variables_initializer())
21                 dae2_W1 = sess.run(self.daes[2].W1)
22                 dae2_b2 = sess.run(self.daes[2].b2)
23         with self.daes[3].tf_graph.as_default():
24             with tf.Session() as sess:
25                 sess.run(tf.global_variables_initializer())
26                 dae3_W1 = sess.run(self.daes[3].W1)
27                 dae3_b2 = sess.run(self.daes[3].b2)
28     # 从 784 维到 1024 维的去噪自动编码器
29     if 'train' == mode:
30         self.W_1 = tf.Variable(dae0_W1, name='W_1')
31         self.b_2 = tf.Variable(dae0_b2, name='b_2')
32     else:
33         self.W_1 = tf.Variable(tf.truncated_normal([784, 1024], mean=0.0,
34             stddev=0.1), name='W_1')
35         self.b_2 = tf.Variable(tf.zeros([1024]), name='b_2')
36     self.z_2 = tf.matmul(self.X, self.W_1) + self.b_2
37     self.a_2 = tf.nn.tanh(self.z_2) # tf.nn.relu(self.z_2)
38     self.a_2_dropout = tf.nn.dropout(self.a_2, self.keep_prob)
39     # 从 1024 维到 784 维的去噪自动编码器
40     if 'train' == mode:
41         self.W_2 = tf.Variable(dae1_W1, name='W_2')
42         self.b_3 = tf.Variable(dae1_b2, name='b_3')
43     else:
44         self.W_2 = tf.Variable(tf.truncated_normal([1024, 784], mean=0.0,
45             stddev=0.1), name='W_2')
46         self.b_3 = tf.Variable(tf.zeros([784]), name='b_3')
47     self.z_3 = tf.matmul(self.a_2_dropout, self.W_2) + self.b_3
48     self.a_3 = tf.nn.tanh(self.z_3) # tf.nn.relu(self.z_3)
49     self.a_3_dropout = tf.nn.dropout(self.a_3, self.keep_prob)
50     # 从 784 维到 512 维的去噪自动编码器
51     if 'train' == mode:
52         self.W_3 = tf.Variable(dae2_W1, name='W_3')
53         self.b_4 = tf.Variable(dae2_b2, name='b_4')
54     else:
55         self.W_3 = tf.Variable(tf.truncated_normal([784, 512], mean=0.0,
56             stddev=0.1), name='W_3')
57         self.b_4 = tf.Variable(tf.zeros([512]), name='b_4')
58     self.z_4 = tf.matmul(self.a_3_dropout, self.W_3) + self.b_4

```

```

59 self.a_4 = tf.nn.tanh(self.z_4) #tf.nn.relu(self.z_4)
60 self.a_4_dropout = tf.nn.dropout(self.a_4, self.keep_prob)
61 # 从 512 维到 256 维的去噪自动编码器
62 if 'train' == mode:
63     self.W_4 = tf.Variable(dae3_W1, name='W_4')
64     self.b_5 = tf.Variable(dae3_b2, name='b_5')
65 else:
66     self.W_4 = tf.Variable(tf.truncated_normal([512, 256], mean=0.0,
67         stddev=0.1), name='W_4')
68     self.b_5 = tf.Variable(tf.zeros([256]), name='b_5')
69 self.z_5 = tf.matmul(self.a_4_dropout, self.W_4) + self.b_5
70 self.a_5 = tf.nn.tanh(self.z_5) #tf.nn.relu(self.z_5)
71 self.a_5_dropout = tf.nn.dropout(self.a_5, self.keep_prob)
72 #输出层
73 self.W_5 = tf.Variable(tf.zeros([256, 10]))
74 self.b_6 = tf.Variable(tf.zeros([10]))
75 self.z_6 = tf.matmul(self.a_5_dropout, self.W_5) + self.b_6
76 self.y_ = tf.nn.softmax(self.z_6)
77 #训练部分
78 self.cross_entropy = tf.reduce_mean(-tf.reduce_sum(
79     self.y * tf.log(self.y_),
80     reduction_indices=[1]))
81 #train_step = tf.train.AdagradOptimizer(0.3).minimize(cross_entropy)
82 self.loss = self.cross_entropy + self.lanmeda*(
83     tf.reduce_sum(self.W_1**2) +
84     tf.reduce_sum(self.W_2**2) + tf.reduce_sum(self.W_3**2) +
85     tf.reduce_sum(self.W_4**2) + tf.reduce_sum(self.W_5**2))
86 self.train_step = tf.train.AdamOptimizer(learning_rate=0.001, beta1=0.9,
87     beta2=0.999, epsilon=1e-08, use_locking=False,
88     name='Adam').minimize(self.loss)
89 self.correct_prediction = tf.equal(tf.argmax(self.y_, 1),
90     tf.argmax(self.y, 1))
91 self.accuracy = tf.reduce_mean(tf.cast(self.correct_prediction,
92     tf.float32))
93 return self.X, self.y_, self.y, self.keep_prob, self.cross_entropy, \
94     self.train_step, self.correct_prediction, self.accuracy

```

第 3 行：定义输入信号的 placeholder。

第 4 行：定义正确标签结果的 placeholder。

第 5 行：采用 Dropout 调整技术时，神经元处于激活状态的比率。

第 6 行：如果创建训练时的模型，需要从已经预训练好的去噪自动编码器中读取参数和超参数；如果创建运行状态的模型，则无须这一步。

第 8 行：打开第一个去噪自动编码器对应的 TensorFlow 的计算图。

第 9 行：开启 TensorFlow 会话。

第 10 行：初始化全局变量。

第 11 行：取出第一个去噪自动编码器的连接权值矩阵。

第 12 行：取出第一个去噪自动编码器的偏置值向量。

第 13 行：打开第二个去噪自动编码器的 TensorFlow 计算图。

第 14 行：开启 TensorFlow 会话。

第 15 行：初始化全局变量。

第 16 行：取出第二个去噪自动编码机的连接权值矩阵。

第 17 行：取出第二个去噪自动编码机的偏置值向量。

第 18 行：打开第三个去噪自动编码机的 TensorFlow 计算图。

第 19 行：打开 TensorFlow 会话。

第 20 行：初始化全局变量。

第 21 行：取出第三个去噪自动编码机的连接权值矩阵。

第 22 行：取出第三个去噪自动编码机的偏置值向量。

第 23 行：打开第四个去噪自动编码机的 TensorFlow 计算图。

第 24 行：开启 TensorFlow 会话。

第 25 行：初始化全局变量。

第 26 行：取出第四个去噪自动编码机的连接权值矩阵。

第 27 行：取出第四个去噪自动编码机的偏置值向量。

第 29~31 行：如果创建训练模型，则将第 1 层到第 2 层的连接权值初始化为第 1 个去噪自动编码机的连接权值，第 2 层的偏置值为第 1 个去噪自动编码机的偏置值。

第 32~35 行：如果创建运行状态的模型，则用均值为 0、标准差为 0.1 的正态分布随机数初始化第 1 层到第 2 层的连接权值矩阵，用 0 初始化第 2 层的偏置值。

第 36 行：求出第 2 层的输入值。

第 37 行：利用双曲正切函数求出第 2 层的原始输出，因为我们的去噪自动编码器采用的是双曲正切激活函数。

第 38 行：采用 Dropout 调整技术，随机选择 `self.keep_prob` 比例的神经元处于激活状态，其余神经元的输出值为 0，得到最终第 2 层的输出值。

第 40~42 行：如果创建训练模型，则将第 2 层到第 3 层的连接权值初始化为第 2 个去噪自动编码机的连接权值，第 3 层的偏置值为第 2 个去噪自动编码机的偏置值。

第 43~46 行：如果创建运行状态的模型，则用均值为 0、标准差为 0.1 的正态分布随机数初始化第 2 层到第 3 层的连接权值矩阵，用 0 初始化第 3 层的偏置值。

第 47 行：求出第 3 层的输入值。

第 48 行：利用双曲正切函数求出第 3 层的原始输出，因为我们的去噪自动编码器采用的是双曲正切激活函数。

第 49 行：采用 Dropout 调整技术，随机选择 `self.keep_prob` 比例的神经元处于激活状态，其余神经元的输出值为 0，得到最终第 3 层的输出值。

第 51~53 行：如果创建训练模型，则将第 3 层到第 4 层的连接权值初始化为第 3 个去噪自动编码机的连接权值，第 4 层的偏置值为第 3 个去噪自动编码机的偏置值。

第 54~57 行：如果创建运行状态的模型，则用均值为 0，标准差为 0.1 的正态分布随机数初始化第 3 层到第 4 层的连接权值矩阵，用 0 初始化第 4 层的偏置值。

第 58 行：求出第 4 层的输入值。

第 59 行：利用双曲正切函数求出第 4 层的原始输出，因为我们的去噪自动编码器采用的是双曲正切激活函数。

第 60 行：采用 Dropout 调整技术，随机选择 `self.keep_prob` 比例的神经元处于激活状态，其余神经元的输出值为 0，得到最终第 4 层的输出值。

第 62~64 行：如果创建训练模型，则将第 4 层到第 5 层的连接权值初始化为第 4 个去噪自动编码机的连接权值，第 5 层的偏置值为第 4 个去噪自动编码机的偏置值。

第 65~68 行：如果创建运行状态的模型，则用均值为 0，标准差为 0.1 的正态分布随机数初始化第 4 层到第 5 层的连接权值矩阵，用 0 初始化第 5 层的偏置值。

第 69 行：求出第 5 层的输入值。

第 70 行：利用双曲正切函数求出第 5 层的原始输出，因为我们的去噪自动编码器采用的是双曲正切激活函数。

第 71 行：采用 Dropout 调整技术，随机选择 `self.keep_prob` 比例的神经元处于激活状态，其余神经元的输出值为 0，得到最终第 5 层的输出值。

第 73 行：定义第 5 层到第 6 层（输出层）的连接权值矩阵，并用全 0 来进行初始化。

第 74 行：定义第 6 层（输出层）的偏置值，并用全 0 来进行初始化。

第 75 行：求出第 6 层的输入值。

第 77 行：利用 Softmax 函数求出第 6 层的输出，即计算出的标签分类结果。

第 78~80 行：定义交叉熵 `cross_entropy` 的计算方法。

第 81 行：定义调整项 L2 权值衰减系数 λ 。

第 82~85 行：定义最终代价函数，值为交叉熵再加上 L2 调整项（权值衰减）。

第 86~88 行：采用梯度下降算法和 Adam 优化算法，利用优化算法求使代价函数达到最小值时连接权值 `W` 和偏移量 `b` 的值。

第 89、90 行：利用 TensorFlow 的 `argmax` 函数，分别求出计算类别向量每个样本的最大值下标和类别向量每个样本的最大值下标，并对其进行比较。

第 91、92 行：求出预测精度，首先调用 TensorFlow 的 `cast` 函数，将第 16 行的结果变为浮点数列列表：`[1.0, 1.0, 0.0, 1.0, 1.0]`，我们这里假设只有 5 个样本。再调用 TensorFlow 的 `reduce_mean` 函数求出这个列表的平均值： $(1.0+1.0+0.0+1.0+1.0)/5=0.8$ ，这个值就是我们模型预测的精度。

第 93、94 行：返回模型定义的 `X, W, b, y_, y, cross_entropy, train_step, correct_prediction, accuracy`。

接下来是训练方法，代码如下：

```
1 def train(self, mode=TRAIN_MODE_NEW, ckpt_file='work/lgr.ckpt'):
2     X_train, y_train, X_validation, y_validation, X_test, \
3         y_test, mnist = self.load_datasets()
4     X, y_, y, keep_prob, cross_entropy, train_step, correct_prediction, \
5         accuracy = self.build_model()
6     epochs = 10
7     saver = tf.train.Saver()
8     total_batch = int(mnist.train.num_examples/self.batch_size)
9     check_interval = 50
10    best_accuracy = -0.01
11    improve_threthold = 1.005
12    no_improve_steps = 0
```

```

13 max_no_improve_steps = 3000
14 is_early_stop = False
15 eval_runs = 0
16 eval_times = []
17 train_accs = []
18 validation_accs = []
19 with tf.Session() as sess:
20     sess.run(tf.global_variables_initializer())
21     if Mlp_Engine.TRAIN_MODE_CONTINUE == mode:
22         saver.restore(sess, ckpt_file)
23     for epoch in range(epochs):
24         if is_early_stop:
25             break
26         for batch_idx in range(total_batch):
27             if no_improve_steps >= max_no_improve_steps:
28                 is_early_stop = True
29                 break
30             X_mb, y_mb = mnist.train.next_batch(self.batch_size)
31             sess.run(train_step, feed_dict={X: X_mb, y: y_mb,
32                 keep_prob: self.keep_prob_val})
33             no_improve_steps += 1
34             if batch_idx % check_interval == 0:
35                 eval_runs += 1
36                 eval_times.append(eval_runs)
37                 train_accuracy = sess.run(accuracy,
38                     feed_dict={X: X_train, y: y_train, keep_prob: 1.0})
39                 train_accs.append(train_accuracy)
40                 validation_accuracy = sess.run(accuracy,
41                     feed_dict={X: X_validation, y: y_validation,
42                         keep_prob: 1.0})
43                 validation_accs.append(validation_accuracy)
44                 if best_accuracy < validation_accuracy:
45                     if validation_accuracy / best_accuracy >= \
46                         improve_threthold:
47                         no_improve_steps = 0
48                         best_accuracy = validation_accuracy
49                         saver.save(sess, ckpt_file)
50                     print('{0}:{1}# train:{2}, validation:{3}'.format(
51                         epoch, batch_idx, train_accuracy,
52                         validation_accuracy))
53             print(sess.run(accuracy, feed_dict={X: X_test,
54                 y: y_test, keep_prob: 1.0}))
55         plt.figure(1)
56         plt.subplot(111)
57         plt.plot(eval_times, train_accs, 'b-', label='train accuracy')
58         plt.plot(eval_times, validation_accs, 'r-',
59             label='validation accuracy')
60         plt.title('accuracy trend')
61         plt.legend(loc='lower right')
62         plt.show()

```

第 2、3 行：读入训练样本集输入信号集、训练样本集标签集、验证样本集输入信号集、验证样本集标签集、测试样本集输入信号集、测试样本集标签集。

第 4、5 行：创建模型，这里创建的模型是 ReLU 神经元模型。

第 6 行：循环学习整个训练样本集遍数。

第 7 行：初始化 TensorFlow 模型保存和恢复对象 `saver`。

第 8 行：用用户训练样本集中样本数除以迷你批次大小，得到迷你批次数量 `total_batch`。

第 9 行：每隔 50 次迷你批次学习，计算在验证样本集上的精度。

第 10 行：保存在验证样本集上所取得的最好的验证样本集精度。

第 11 行：定义验证样本集上精度提高 0.5% 时才算显著提高。

第 12 行：记录在验证样本集上精度没有显著提高学习迷你批次的次数。

第 13 行：在验证样本集精度最大没有显著提高的情况下，允许学习迷你批次的次数。

第 14 行：是否终止学习过程。

第 15 行：评估验证样本集上识别精度的次数。

第 16 行：用 `eval_times` 列表来保存评估次数，作为后面绘制识别精度趋势图的横坐标。

第 17 行：用 `train_accs` 列表保存在训练样本集上每次评估时的识别精度，作为后面图形深色曲线的纵坐标。

第 18 行：用 `validation_accs` 列表保存在验证样本集上每次评估时的识别精度，作为后面图形浅色曲线的纵坐标。

第 19 行：启动 TensorFlow 会话。

第 20 行：初始化全局参数。

第 21、22 行：如果模式为 `TRAIN_MODE_CONTINUE`，则读入以前保存的 `ckpt` 模型文件，初始化模型参数。

第 23 行：循环第 20~52 行操作，对整个训练样本集进行一次学习。

第 24、25 行：如果 `is_early_stop` 为真，则终止本层循环。

第 26 行：循环第 23~52 行操作，对一个迷你批次进行学习。

第 27~29 行：如果验证样本集上识别精度没有显著改善的迷你批次学习次数大于最大允许的验证样本集上识别精度没有显著改善的迷你批次学习次数，则将 `is_early_stop` 置为真，并退出本层循环。这会直接触发第 24、25 行操作，终止外层循环，学习过程结束。

第 30 行：从训练样本集中取出一个迷你批次的输入信号集 `X_mb` 和标签集 `y_mb`。

第 31、32 行：调用 TensorFlow 计算模型输出、代价函数，求出代价函数对参数的导数，并应用梯度下降算法更新模型参数值。注意，此时我们采用 Dropout 调整技术，将隐藏层神经元保留比例作为一个参数 `keep_prob` 传给模型。

第 33 行：将验证样本集没有显著改善的迷你批次学习次数加 1。

第 34 行：如果连续进行了指定次数的迷你批次学习，则计算统计信息。

第 35 行：识别精度评估次数加 1。

第 36 行：将识别精度评估次数加入 `eval_times`（图形横坐标）列表中。

第 37、38 行：计算训练样本集上的识别精度。

第 39 行：将训练样本集上的识别精度加入训练样本集识别精度列表 `train_accs` 中。

第 40~42 行：计算验证样本集上的识别精度。注意，此时我们采用 Dropout 调整技术，将隐藏层神经元保留比例作为一个参数 `keep_prob` 传给模型。这里我们给出的是 1.0，即全部保留。这是一个惯例，即在训练时使用 Dropout 调整技术，在评估和运行时不使用 dropout

调整技术，读者一定要注意。

第 43 行：将验证样本集上的识别精度加入到验证样本集识别精度列表 `validation_accs` 中。

第 44 行：如果验证样本集上最佳识别精度小于当前的验证样本集上的识别精度，执行第 45~49 行操作。

第 45~47 行：如果当前验证样本集上的识别精度比之前的最佳识别精度提高 0.5% 以上，则将验证样本集没有显著改善的迷你批次学习次数设为 0。

第 48 行：将验证样本集上最佳识别精度的值设置为当前验证样本集上识别精度值。

第 49 行：将当前模型参数保存到 `ckpt` 模型文件中。

第 50~52 行：打印训练状态信息。

第 53、54 行：训练完成后，计算测试样本集上的识别精度，并打印出来。

第 55、56 行：初始化 `matplotlib` 绘图库。

第 57 行：绘制训练样本集上识别精度的变化趋势曲线，用蓝色绘制。

第 58、59 行：绘制验证样本集上识别精度的变化趋势曲线，用红色绘制。

第 60 行：设置图形标题。

第 61 行：在右下角添加图例。

第 62 行：具体绘制图像。

运行深度信念网络训练方法，会有如图 11.2 所示的结果输出。

```
Build RBM Model
0_0:cost=0.8682435750961304 <class 'numpy.ndarray', (512, 256)
0_1:cost=0.8666553497314453 <class 'numpy.ndarray', (512, 256)
0_2:cost=0.8649016618728638 <class 'numpy.ndarray', (512, 256)
0_3:cost=0.8628353476524353 <class 'numpy.ndarray', (512, 256)
0_4:cost=0.8613502383232117 <class 'numpy.ndarray', (512, 256)
0_5:cost=0.859567403793335 <class 'numpy.ndarray', (512, 256)
0_6:cost=0.8578780889511108 <class 'numpy.ndarray', (512, 256)
0_7:cost=0.8563374280929565 <class 'numpy.ndarray', (512, 256)
0_8:cost=0.8546863794326782 <class 'numpy.ndarray', (512, 256)
0_9:cost=0.8531083464622498 <class 'numpy.ndarray', (512, 256)
0_10:cost=0.8512935638427734 <class 'numpy.ndarray', (512, 256)
0_11:cost=0.8496608734130859 <class 'numpy.ndarray', (512, 256)
0_12:cost=0.8479791879653931 <class 'numpy.ndarray', (512, 256)
0_13:cost=0.8465133905410767 <class 'numpy.ndarray', (512, 256)
0_14:cost=0.8450098633766174 <class 'numpy.ndarray', (512, 256)
0_15:cost=0.8435034155845642 <class 'numpy.ndarray', (512, 256)
0_16:cost=0.8417938947677612 <class 'numpy.ndarray', (512, 256)
0_17:cost=0.8401952385902405 <class 'numpy.ndarray', (512, 256)
0_18:cost=0.8388243317604065 <class 'numpy.ndarray', (512, 256)
0_19:cost=0.8374171853065491 <class 'numpy.ndarray', (512, 256)
0_20:cost=0.835780143737793 <class 'numpy.ndarray', (512, 256)
0_21:cost=0.8343423008918762 <class 'numpy.ndarray', (512, 256)
0_22:cost=0.8329617381095886 <class 'numpy.ndarray', (512, 256)
0_23:cost=0.8315491080284119 <class 'numpy.ndarray', (512, 256)
0_24:cost=0.8301364779472351 <class 'numpy.ndarray', (512, 256)
0_25:cost=0.8286508917808533 <class 'numpy.ndarray', (512, 256)
0_26:cost=0.8272291421890259 <class 'numpy.ndarray', (512, 256)
0_27:cost=0.8259280323982239 <class 'numpy.ndarray', (512, 256)
0_28:cost=0.8244368433952332 <class 'numpy.ndarray', (512, 256)
0_29:cost=0.8230018615722656 <class 'numpy.ndarray', (512, 256)
0_30:cost=0.8217670917510986 <class 'numpy.ndarray', (512, 256)
```

图 11.2 预训练阶段输出结果

上图为逐层预训练受限玻尔兹曼机时后台输出的训练结果信息，由于内容很多，我们只截取了某个受限玻尔兹曼机训练开始时的输出。调优阶段的输出结果如图 11.3 所示。

```

5:0# train:0.9402363896369934, validation:0.9419999718666077
5:50# train:0.9421818256378174, validation:0.9444000124931335
5:100# train:0.9467636346817017, validation:0.9440000057220459
5:150# train:0.9409090876579285, validation:0.9441999793052673
5:200# train:0.9341272711753845, validation:0.9359999895095825
5:250# train:0.9415454268455505, validation:0.9423999786376953
5:300# train:0.9401454329490662, validation:0.9423999786376953
5:350# train:0.9479818344116211, validation:0.9517999887466431
5:400# train:0.9480000138282776, validation:0.9484000205993652
5:450# train:0.9412363767623901, validation:0.9409999847412109
5:500# train:0.9477999806404114, validation:0.9527999758720398
6:0# train:0.9478545188903809, validation:0.9455999732017517
6:50# train:0.9470363855361938, validation:0.9520000219345093
6:100# train:0.9394727349281311, validation:0.9473999738693237

```

图 11.3 调优阶段输出结果

调优阶段的深度信念网络相当于一个完整的多层感知器模型，其训练过程也和我们之前讲解的多层感知器模型的训练过程非常类似。

图 11.4 为调优阶段在训练样本集上的识别精度和在验证样本集上的识别精度的变化趋势图。由图可见，这两者之间没有出现明显的过拟合现象，因此通过调整调整项，如 Early Stopping 标准等，网络的识别精度还可以有一定的提升空间。

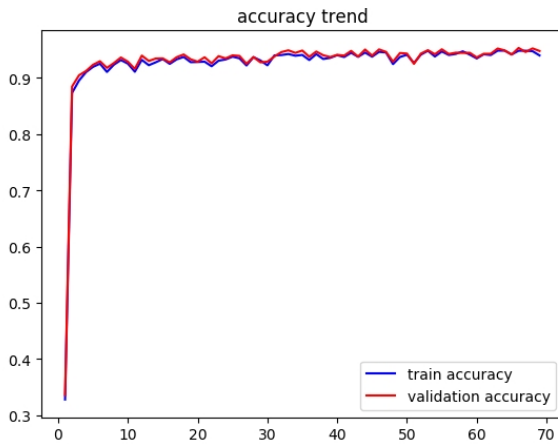


图 11.4 调优阶段识别精度变化趋势图

在深度信念网络训练完成之后，就可以进入运行状态。在运行状态下，深度信念网络实际上就是一个多层感知器模型，代码如下：

```

1 def run(self, ckpt_file='work/lgr.ckpt'):
2     print('run.....')
3     img_file = 'datasets/test5.png'
4     img = io.imread(img_file, as_grey=True)
5     raw = [1 if x<0.5 else 0 for x in img.reshape(784)]
6     #sample = np.array(raw)
7     X_train, y_train, X_validation, y_validation, \
8         X_test, y_test, mnist = self.load_datasets()
9     sample = X_test[102]
10    X_run = sample.reshape(1, 784)
11    digit = -1
12    with tf.Graph().as_default():
13        X, y_, y, keep_prob, cross_entropy, train_step, correct_prediction, \
14            accuracy = self.build_model(mode='run')
15        saver = tf.train.Saver()

```

```

16     with tf.Session() as sess:
17         sess.run(tf.global_variables_initializer())
18         saver.restore(sess, ckpt_file)
19         rst = sess.run(y_, feed_dict={X: X_run, keep_prob: 1.0})
20         print('rst:{0}'.format(rst))
21         max_prob = -0.1
22         for idx in range(10):
23             if max_prob < rst[0][idx]:
24                 max_prob = rst[0][idx]
25                 digit = idx
26         # W_1_1
27         W_1 = sess.run(self.W_1)
28         wight_map = W_1[:,0].reshape(28, 28)
29         a_2 = sess.run(self.a_2, feed_dict={X: X_run, \
30             keep_prob: 1.0})
31         a_2_raw = a_2[0]
32         a_2_img = a_2_raw[0:784]
33         feature_map = a_2_img.reshape(28, 28)
34     img_in = sample.reshape(28, 28)
35     plt.figure(1)
36     plt.subplot(131)
37     plt.imshow(img_in, cmap='gray')
38     plt.title('result:{0}'.format(digit))
39     plt.axis('off')
40     plt.subplot(132)
41     plt.imshow(wight_map, cmap='gray')
42     plt.axis('off')
43     plt.title('wight row')
44     plt.subplot(133)
45     plt.imshow(feature_map, cmap='gray')
46     plt.axis('off')
47     plt.title('hidden layer')
48     plt.show()

```

第 3 行：用绘图软件做一个 28×28 的图像，在上面写一个数字，这里我们直接写上一个印刷体的 5。

第 4 行：以灰度图像方式读出图像内容。

第 5 行：首先将其形状从 28×28 二维变为一维 784，然后根据每个像素的值进行处理：值小于 0.5 取 1，否则取 0。

第 6 行：将其变为 numpy 数组，作为一个样本。

第 7、8 行：调用 load_datasets 方法，读入训练样本集、验证样本集、测试样本集的内容。

第 9 行：我们取测试样本集中第 103 个样本作为测试样本。我们的模型在训练过程中没有见过测试样本集中的样本，因此可以模拟实际应用中遇到的情况。

第 10 行：将其变为[1, 784]矩阵形式，我们可以称之为运行样本集，其中只有一个样本。

第 11 行：定义 digit 为最终识别出的 0~9 的数字，取-1 表示还没有识别结果。

第 12 行：打开 TensorFlow 的计算图。

第 13、14 行：以运行模式创建模型（此时无须读入预训练的去噪自动编码器的参数和超参数）。

第 15 行：初始化 TensorFlow 的 saver 对象。

第 16 行：启动 TensorFlow 会话来运行程序。

第 17 行：初始化变量。

第 18 行：恢复之前保存的模型参数文件，并初始化模型参数。

第 19 行：求以样本 `X_run` 为输入，在输出层经过 `Softmax` 函数后的计算值，表示每个类别的概率。注意：这里设置 `Dropout` 技术中隐藏层神经元的保留率为 1.0，即所有隐藏层神经元均参与运算，相当于多个随机网络共同投票得出最终结果。

第 20 行：打印出所有类别的概率值。

第 21 行：定义 `max_prob` 记录所有类别最大的概率值。

第 22 行：循环识别结果 `rst` 每个类别的概率。

第 23~25 行：如果最大概率小于当前类别的概率，将当前概率赋给最大概率，识别出的数字等于类别索引值，即对应的 0~9 中的数字。当循环完所有类别后，我们就能找到概率最大的类别及其对应的数字了。

第 27 行：取出输入层到隐藏层的连接权值矩阵 `W_1`（实际为其转置）。

第 28 行：将第一行形状转为 28×28 的图片数据格式。

第 29、30 行：求出隐藏层神经元输出值。注意：这里设置 `Dropout` 技术中隐藏层神经元的保留率为 1.0，即所有隐藏层神经元均参与运算，相当于多个随机网络共同投票得出最终结果。

第 31 行：取出隐藏层输出的原始数据。

第 32 行：由于隐藏层输出数据为 1024 维，我们想显示成一个正方形数据，所以我们只取前面的 784（28×28）维数据。

第 33 行：将隐藏层输出前 784 维变为 28×28 的特征图。

第 34 行：将输入样本变为 28×28 的黑白图片。

第 35、36 行：初始化 `matplotlib` 绘图库。

第 37~39 行：绘制识别的输入图像，将识别结果显示在标题上。

第 40~43 行：显示输入层到隐藏层第一行连接权值的图像。

第 44~47 行：显示隐藏层输出值图像，因为多层感知器是将原始输入信号进行变换，变为适合分类识别的形式，可以视为一种特征学习形式。

第 48 行：同时显示三幅图像。

运行上面的程序，输出结果如图 11.5 所示。

```
rst:[[ 3.04061570e-04  9.61051501e-06  3.73906505e-06  1.02313503e-03
       7.19604850e-06  9.96774733e-01  9.46061959e-07  6.87801075e-05
       1.15693687e-03  6.50937087e-04]]
```

图 11.5 深度信念网络运行结果

由上图可以看出，在 5 处概率值最大，因此网络判断该样本类别为 5，这与实际情况相符合。

如图 11.6 所示，原始输入样本为左边第一个图片，中间的图片为第一层连接权值截取出的 28×28 个数据绘制的图像，最右侧的图片为第一个隐藏层截取出的 28×28 的数据组成的图像。由图可以看出，深度信念网络的行为表现与标准的多层感知器模型基本一致，唯一不同的是深度信念网络由于逐层预训练的存在，可以更快地达到收敛状态，并且可能拥有更好的识别精度。

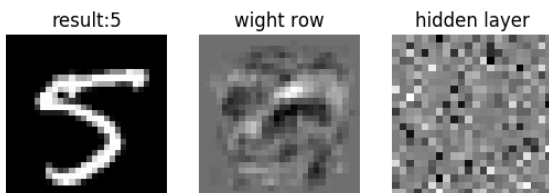


图 11.6 深度信念网络中的输入值、权值、隐藏层

11.3 深度信念网络 Theano 实现

下面来看怎样通过 Theano 框架实现深度信念网络。其实深度信念网络与堆叠去噪自动编码器类似，只是堆叠去噪自动编码器使用的是去噪自动编码器，而这里我们将使用受限玻尔兹曼机。读者可以将两部分内容结合起来看。

首先定义 DBN 类，代码如下：

```
1 from __future__ import print_function, division
2 import os
3 import sys
4 import timeit
5 import numpy
6 import theano
7 import theano.tensor as T
8 from theano.sandbox.rng_mrg import MRG_RandomStreams
9 from logistic_sgd import LogisticRegression, load_data
10 from mlp import HiddenLayer
11 from rbm import RBM
12
13
14 class DBN(object):
15     """
16     深度信念网络 (DBN) 是由多个堆叠起来的受限玻尔兹曼机 (RBM) 组成。在第 i 层
17     RBM 的隐藏层将作为第 i+1 层的输入层。第 1 层 RBM 可见层加载网络的输入信号，
18     而最后一个 RBM 的隐藏层作为整个网络的输出。当用于模式分类应用时，深度信念
19     网络 (DBN) 可以采用与多层感知器 (MLP) 相类似的训练方法，并且在最上层加
20     一个逻辑回归层 (多类时为 softmax 回归) 来实现。
21     """
22
23     def __init__(self, numpy_rng, theano_rng=None, n_ins=784,
24                 hidden_layers_sizes=[500, 500], n_outs=10):
25         """This class is made to support a variable number of layers.
26         通过 hidden_layers_sizes，本类可以支持建立任意多层。
27         参数
28         - numpy_rng : numpy 随机数生成引擎
29         - theano_rng : Theano 随机数生成引擎
30         - n_ins : 输入层神经元数，本例中为  $28 \times 28 = 784$ 
31         - hidden_layers_sizes : 各隐藏层神经元数大小，这里指定最终的层数
32         - n_outs : 输出层神经元数，即分类类别数
33         """
```



```

34     self.sigmoid_layers = []
35     self.rbm_layers = []
36     self.params = []
37     self.n_layers = len(hidden_layers_sizes)
38     assert self.n_layers > 0
39
40     if not theano_rng:
41         theano_rng = MRG_RandomStreams(numpy_rng.randint(2 ** 30))
42
43     # allocate symbolic variables for the data
44     # the data is presented as rasterized images
45     self.x = T.matrix('x')
46
47     # the labels are presented as 1D vector of [int] labels
48     self.y = T.ivector('y')
49     # end-snippet-1
50     # The DBN is an MLP, for which all weights of intermediate
51     # layers are shared with a different RBM. We will first
52     # construct the DBN as a deep multilayer perceptron, and when
53     # constructing each sigmoidal layer we also construct an RBM
54     # that shares weights with that layer. During pretraining we
55     # will train these RBMs (which will lead to changing the
56     # weights of the MLP as well) During finetuning we will finish
57     # training the DBN by doing stochastic gradient descent on the
58     # MLP.
59
60     for i in range(self.n_layers):
61         # construct the sigmoidal layer
62         # the size of the input is either the number of hidden
63         # units of the layer below or the input size if we are on
64         # the first layer
65         if i == 0:
66             input_size = n_ins
67         else:
68             input_size = hidden_layers_sizes[i - 1]
69
70         # the input to this layer is either the activation of the
71         # hidden layer below or the input of the DBN if you are on
72         # the first layer
73         if i == 0:
74             layer_input = self.x
75         else:
76             layer_input = self.sigmoid_layers[-1].output
77
78         sigmoid_layer = HiddenLayer(rng=numpy_rng,
79                                     input=layer_input,
80                                     n_in=input_size,
81                                     n_out=hidden_layers_sizes[i],
82                                     activation=T.nnet.sigmoid)
83
84         # add the layer to our list of layers
85         self.sigmoid_layers.append(sigmoid_layer)
86
87         # its arguably a philosophical question... but we are
88         # going to only declare that the parameters of the
89         # sigmoid layers are parameters of the DBN. The visible
90         # biases in the RBM are parameters of those RBMs, but not
91         # of the DBN.
92         self.params.extend(sigmoid_layer.params)
93
94         # Construct an RBM that shared weights with this layer
95         rbm_layer = RBM(numpy_rng=numpy_rng,
96                         theano_rng=theano_rng,
97                         input=layer_input,
98                         n_visible=input_size,
99                         n_hidden=hidden_layers_sizes[i],
100                         W=sigmoid_layer.W,
101                         hbias=sigmoid_layer.b)
102         self.rbm_layers.append(rbm_layer)
103
104     # We now need to add a logistic layer on top of the MLP
105     self.logLayer = LogisticRegression(
106         input=self.sigmoid_layers[-1].output,
107         n_in=hidden_layers_sizes[-1],
108         n_out=n_outs)

```

```

109         self.params.extend(self.logLayer.params)
110
111         # compute the cost for second phase of training, defined as the
112         # negative log likelihood of the logistic regression (output) layer
113         self.finetune_cost = self.logLayer.negative_log_likelihood(self.y)
114
115         # compute the gradients with respect to the model parameters
116         # symbolic variable that points to the number of errors made on the
117         # minibatch given by self.x and self.y
118         self.errors = self.logLayer.errors(self.y)

```

第 34 行：定义属性 `sigmoid_layers`，代表多层感知器列表。因为深度信念网络的网络既可以被视为多层感知器，也可以被视为由多个受限玻尔兹曼机堆叠而成。

第 35 行：定义属性 `rbm_layers`，其与 `sigmoid_layers` 是同样的网络。

第 36 行：定义属性 `params`，维护模型中的参数。

第 37 行：定义属性 `n_layers`，表示网络由多少层组成，层数是由参数 `hidden_layers_sizes` 指定的。

第 40、41 行：初始化 Theano 随机数生成引擎。

第 45 行：定义属性 `x` 为定义矩阵，每一行代表一个样本，所有行组成一个迷你批次量。

第 48 行：定义属性 `y` 为输出标签向量，在本例中为维度为 10 的向量，只有一项可以为 1，表示是 0~9 中的哪一个。

第 60 行：对深度信念网络中的每一层进行循环。深度信念网络可以被视为多层感知器，每层可以和其上面一层共同组成一个受限玻尔兹曼机，这两种类型的网络共享神经元和连接权值矩阵。所以在下面的代码中，我们虽然创建的是多层感知器的层，实际上就是受限玻尔兹曼机。我们在下一步将要进行的预训练，就是要操作受限玻尔兹曼机。

第 65、66 行：当为第一层时，输入信号维度为网络的输入信号维度。

第 67、68 行：当不为第一层时，输入信号维度为前一层网络的神经元数量。

第 73、74 行：如果是第一层时，输入信号为网络的输入信号。

第 75、76 行：如果不是第一层时，输入信号为前一层的输出值。

第 78~82 行：创建隐藏层实例（实际上也是 RBM 网络的一层）。

❑ `rng`: numpy 随机数生成引擎。

❑ `input`: 输入信号。

❑ `n_in`: 输入信号维度。

❑ `n_out`: 输出信号维度，本层神经元数量。

❑ `activation`: 激活函数，在本例中选择的是 Sigmoid 函数，实际上，若将其替换为双曲正切函数或 ReLU 神经元，可以取得更好的效果。

第 85 行：将新建立的隐藏层添加到 `sigmoid_layers` 列表中。

第 92 行：将隐藏层的参数添加到网络参数中。注意：这里我们只添加了隐藏层的参数，当该层作为受限玻尔兹曼机可见层时，可见层神经元偏移量不包括在其中，将在下面与此层相对应的 RBM 层中进行定义。

第 95~101 行：定义 RBM 层实例。

❑ `numpy_rng`: numpy 随机数生成引擎。

❑ `theano_rng`: theano 随机数生成引擎。

- ❑ **input**: 输入信号为本层的输入信号。
- ❑ **n_visible**: 可见层神经元数为本层输入信号维度。
- ❑ **n_hidden**: 隐藏层神经元数为本层神经元数。
- ❑ **W**: 连接权值矩阵与多层感知器对应的隐藏层共享。
- ❑ **hbias**: 隐藏层偏移量与多层感知器对应的隐藏层神经元偏移量共享。

第 102 行: 将新创建的 RBM 层加入 `rbm_layers` 列表中。

第 105~108 行: 在网络最顶层添加了一个逻辑回归层, 定义属性 `logLayer`。

- ❑ **input**: 输入信号为多层感知器最后一层的输出。
- ❑ **n_in**: 输入信号维度为多层感知器最后一层的神经元数量。
- ❑ **n_out**: 整个网络的输出类别。

第 109 行: 将逻辑回归层参数添加到模型参数中。

第 113 行: 定义属性 `finetune_cost` 为整个网络的代价函数, 其定义为逻辑回归层定义的负对数似然函数。

第 118 行: 将整个模型的误差定义为逻辑回归层的误差计算函数。

在定义完整个网络之后, 我们将开始预训练过程, 即对每个 RBM 网络分别进行非监督学习训练, 代码如下:

```
120 def pretraining_functions(self, train_set_x, batch_size, k):
121     '''
122     返回每个受限波尔兹曼机的训练函数的列表, 第i个元素代表第i个
123     受限波尔兹曼机的训练函数
124     参数
125     - train_set_x: 训练样本集输入集
126     - batch_size: 迷你批次中样本数
127     - k: CD-K或PCD-K中的K值
128     '''
129     # index to a [mini]batch
130     index = T.lscalar('index') # index to a minibatch
131     learning_rate = T.scalar('lr') # learning rate to use
132
133     # beginning of a batch, given `index`
134     batch_begin = index * batch_size
135     # ending of a batch given `index`
136     batch_end = batch_begin + batch_size
137
138     pretrain_fns = []
139     for rbm in self.rbm_layers:
140
141         # get the cost and the updates list
142         # using CD-k here (persistent=None) for training each RBM.
143         # TODO: change cost function to reconstruction error
144         cost, updates = rbm.get_cost_updates(learning_rate,
145                                             persistent=None, k=k)
146
147         # compile the theano function
148         fn = theano.function(
149             inputs=[index, theano.In(learning_rate, value=0.1)],
150             outputs=cost,
151             updates=updates,
152             givens={
153                 self.x: train_set_x[batch_begin:batch_end]
154             }
155         )
156         # append `fn` to the list of functions
157         pretrain_fns.append(fn)
158
159     return pretrain_fns
```

第 130 行: 定义 `index` 为迷你批次索引号。

第 131 行: 定义学习率为 Theano 变量 `learning_rate`。

第 134 行：计算迷你批次的开始索引号 `batch_begin`。

第 136 行：计算迷你批次的结束索引号 `batch_end`。

第 138 行：定义预训练函数列表 `pretrain_fns`，每一项对应一个 RBM 网络的预训练函数。

第 139 行：对模型中的每个 RBM 网络进行循环。

第 144、145 行：定义 RBM 网络代价函数计算函数和参数更新规则。

第 148~155 行：编译成 Theano 函数。

❑ `input`：输入参数包括迷你批次中的索引号和学习率。

❑ `outputs`：代价函数值。

❑ `updates`：参数更新情况。

❑ `givens`：取一个迷你批次作为参数。

第 157 行：将定义好的 Theano 函数加入预训练函数列表中。

第 159 行：返回预训练函数列表。

下面来看微调阶段训练函数的定义，代码如下：

```
161 def build_finetune_functions(self, datasets, batch_size, learning_rate):
162     """
163     生成微调网络的训练函数train，执行一步微调动作。定义validate函数，
164     求出在验证样本集上的误差，定义test函数计算在测试样本集上的误差
165     参数
166     - datasets：数据集，包括训练样本集、验证样本集、测试样本集，每个样本
167       集中包括输入集和标签集
168     - batch_size：迷你批次中样本数
169     - learning_rate：学习率
170     """
171     (train_set_x, train_set_y) = datasets[0]
172     (valid_set_x, valid_set_y) = datasets[1]
173     (test_set_x, test_set_y) = datasets[2]
174
175     # compute number of minibatches for training, validation and testing
176     n_valid_batches = valid_set_x.get_value(borrow=True).shape[0]
177     n_valid_batches //= batch_size
178     n_test_batches = test_set_x.get_value(borrow=True).shape[0]
179     n_test_batches //= batch_size
180
181     index = T.lscalar('index') # index to a [mini]batch
182
183     # compute the gradients with respect to the model parameters
184     gparams = T.grad(self.finetune_cost, self.params)
185
186     # compute list of fine-tuning updates
187     updates = []
188     for param, gparam in zip(self.params, gparams):
189         updates.append((param, param - gparam * learning_rate))
190
191     train_fn = theano.function(
192         inputs=[index],
193         outputs=self.finetune_cost,
194         updates=updates,
195         givens={
196             self.x: train_set_x[
197                 index * batch_size: (index + 1) * batch_size
198             ],
199             self.y: train_set_y[
200                 index * batch_size: (index + 1) * batch_size
201             ]
202         }
203     )
204
205     test_score_i = theano.function(
206         [index],
207         self.errors,
208         givens={
209             self.x: test_set_x[
```

```

210         index * batch_size: (index + 1) * batch_size
211     ],
212     self.y: test_set_y[
213         index * batch_size: (index + 1) * batch_size
214     ]
215 }
216 )
217
218 valid_score_i = theano.function(
219     [index],
220     self.errors,
221     givens={
222         self.x: valid_set_x[
223             index * batch_size: (index + 1) * batch_size
224         ],
225         self.y: valid_set_y[
226             index * batch_size: (index + 1) * batch_size
227         ]
228     }
229 )
230
231 # Create a function that scans the entire validation set
232 def valid_score():
233     return [valid_score_i(i) for i in range(n_valid_batches)]
234
235 # Create a function that scans the entire test set
236 def test_score():
237     return [test_score_i(i) for i in range(n_test_batches)]
238
239 return train_fn, valid_score, test_score

```

第 171 行：从数据集的训练样本集中取出训练输入信号集和训练输出标签集。

第 172 行：从数据集的验证样本集中取出验证输入信号集和验证输出标签集。

第 173 行：从数据集的测试样本集中取出测试输入信号集和测试输出标签集。

第 176、177 行：求出验证样本集中的迷你批次数。

第 178、179 行：求出测试样本集中的迷你批次数。

第 181 行：定义 `index` 为迷你批次中的索引号。

第 184 行：求出所有参数对微调网络代价函数的导数。

第 187 行：定义 `updates` 保存所有参数的更新值。

第 188、189 行：对每个参数按照以下公式计算更新值： $\text{param} = \text{param} - \text{learning_rate} \times \text{grad}$ ，其中 `grad` 为该参数对代价函数的导数。

第 191~203 行：定义 Theano 函数 `train_fn` 为训练函数，其输入为训练样本集一个迷你批次，输出为微调网络的代价函数值，更新为上一步求出的所有参数的更新值。

第 205~216 行：定义 Theano 函数求测试样本集在指定迷你批次上的分数，输入为测试样本集上一个迷你批次，输出为误差函数计算的误差。

第 218~229 行：定义 Theano 函数求验证样本集指定迷你批次上的分数，输入为验证样本集上一个迷你批次，输出为误差函数计算的误差。

第 232~233 行：定义 `valid_score` 函数，计算验证样本集上所有迷你批次的分数。

第 236、237 行：定义 `test_score` 函数，计算测试样本集上所有迷你批次的分数。

下面来看深度信念网络的训练过程，代码如下：

```

242 def test_DBN(finetune_lr=0.1, pretraining_epochs=100,
243               pretrain_lr=0.01, k=1, training_epochs=1000,
244               dataset='mnist.pkl.gz', batch_size=10):
245     """
246     以MNIST手写数字识别为例，演示怎样训练和测试深度信念网络
247     参数
248     - finetune_lr：微调网络学习率
249     - pretraining_epochs：预训练训练样本集遍历次数
250     - pretrain_lr：预训练学习率
251     - k：CD-K或PCD-K算法中的K值
252     - training_epochs：微调网络训练样本集遍历次数
253     - dataset：数据集
254     - batch_size：迷你批次中样本数
255     """
256     datasets = load_data(dataset)
257
258     train_set_x, train_set_y = datasets[0]
259     valid_set_x, valid_set_y = datasets[1]
260     test_set_x, test_set_y = datasets[2]
261
262     # compute number of minibatches for training, validation and testing
263     n_train_batches = train_set_x.get_value(borrow=True).shape[0] // batch_size
264
265     # numpy random generator
266     numpy_rng = numpy.random.RandomState(123)
267     print('... building the model')
268     # construct the Deep Belief Network
269     dbn = DBN(numpy_rng=numpy_rng, n_ins=28 * 28,
270               hidden_layers_sizes=[1000, 1000, 1000],
271               n_outs=10)
272
273     # start-snippet-2
274     #####
275     # PRETRAINING THE MODEL #
276     #####
277     print('... getting the pretraining functions')
278     pretraining_fns = dbn.pretraining_functions(train_set_x=train_set_x,
279                                                batch_size=batch_size,
280                                                k=k)
281
282     print('... pre-training the model')
283     start_time = timeit.default_timer()
284     # Pre-train layer-wise
285     for i in range(dbn.n_layers):
286         # go through pretraining epochs
287         for epoch in range(pretraining_epochs):
288             # go through the training set
289             c = []
290             for batch_index in range(n_train_batches):
291                 c.append(pretraining_fns[i](index=batch_index,
292                                           lr=pretrain_lr))
293             print('Pre-training layer %i, epoch %d, cost ' % (i, epoch), \
294                   end=' ')
295             print(numpy.mean(c, dtype='float64'))
296
297     end_time = timeit.default_timer()
298     # end-snippet-2
299     print('The pretraining code for file ' + os.path.split(__file__)[1] +
300           ' ran for %.2fm' % ((end_time - start_time) / 60.), file=sys.stderr)
301     #####
302     # FINETUNING THE MODEL #
303     #####
304
305     # get the training, validation and testing function for the model
306     print('... getting the finetuning functions')
307     train_fn, validate_model, test_model = dbn.build_finetune_functions(
308         datasets=datasets,
309         batch_size=batch_size,
310         learning_rate=finetune_lr
311     )
312
313     print('... finetuning the model')
314     # early-stopping parameters
315
316     # look as this many examples regardless
317     patience = 4 * n_train_batches

```

```

318
319     # wait this much longer when a new best is found
320     patience_increase = 2.
321
322     # a relative improvement of this much is considered significant
323     improvement_threshold = 0.995
324
325     # go through this many minibatches before checking the network on
326     # the validation set; in this case we check every epoch
327     validation_frequency = min(n_train_batches, patience / 2)
328
329     best_validation_loss = numpy.inf
330     test_score = 0.
331     start_time = timeit.default_timer()
332
333     done_looping = False
334     epoch = 0
335
336     while (epoch < training_epochs) and (not done_looping):
337         epoch = epoch + 1
338         for minibatch_index in range(n_train_batches):
339
340             train_fn(minibatch_index)
341             iter = (epoch - 1) * n_train_batches + minibatch_index
342
343             if (iter + 1) % validation_frequency == 0:
344
345                 validation_losses = validate_model()
346                 this_validation_loss = numpy.mean(validation_losses, \
347                                         dtype='float64')
348                 print('epoch %i, minibatch %i/%i, validation error %f %%' % (
349                     epoch,
350                     minibatch_index + 1,
351                     n_train_batches,
352                     this_validation_loss * 100.
353                 )
354             )
355
356             # if we got the best validation score until now
357             if this_validation_loss < best_validation_loss:
358
359                 # improve patience if loss improvement is good enough
360                 if (this_validation_loss < best_validation_loss *
361                     improvement_threshold):
362                     patience = max(patience, iter * patience_increase)
363
364                 # save best validation score and iteration number
365                 best_validation_loss = this_validation_loss
366                 best_iter = iter
367
368                 # test it on the test set
369                 test_losses = test_model()
370                 test_score = numpy.mean(test_losses, dtype='float64')
371                 print(('    epoch %i, minibatch %i/%i, test error of '
372                     'best model %f %%') %
373                     (epoch, minibatch_index + 1, n_train_batches,
374                     test_score * 100.))
375
376             if patience <= iter:
377                 done_looping = True
378                 break
379
380     end_time = timeit.default_timer()
381     print(('Optimization complete with best validation score of %f %%,'
382           'obtained at iteration %i,'
383           'with test performance %f %%'
384           ) % (best_validation_loss * 100., best_iter + 1, test_score * 100.))
385     print('The fine tuning code for file ' + os.path.split(__file__)[1] +
386           ' ran for %.2fm' % ((end_time - start_time) / 60.), file=sys.stderr)

```

第 256 行：读入 MNIST 手写数字识别数据集。

第 258 行：从数据集的训练样本集中读出训练输入信号集和训练输出标签集。

第 259 行：从数据集的验证样本集中读出验证输入信号集和验证输出标签集。

第 260 行：从数据集的测试样本集中读出测试输入信号集和测试输出标签集。

第 263 行：求出训练样本集中迷你批次数量。

第 266 行：初始化 `numpy` 随机数生成引擎。

第 269~270 行：生成深度信念网络类实例。

❑ `numpy_rng`: `numpy` 随机数生成引擎。

❑ `n_ins`: 输入信号维度，此处为黑白图像的分辨率 28×28。

❑ `hidden_layers_sizes`: 隐藏层数量和每层神经元数量。

❑ `n_outs`: 输出层类别数。

第 278~280 行：获取受限玻尔兹曼机的预训练函数，参数为：训练样本集中输入信号集、迷你批次中样本数、CD-K 或 PCD-K 算法中的 K 值。

第 283 行：记录预训练开始时间。

第 285 行：对所有组成的受限玻尔兹曼机进行循环。

第 287 行：对每个受限玻尔兹曼机执行 `pretraining_epochs` 次预训练。

第 288、289 行：定义变量 `c` 记录训练过程中代价函数值变化的列表。

第 290 行：对每个迷你批次进行循环。

第 291、292 行：调用受限玻尔兹曼机的预训练函数，计算代价函数值并更新参数值。

第 293~295 行：打印本次训练汇总信息。注意：这里的代价函数值为所有迷你批次代价函数值的平均值。

第 297 行：记录预训练结束时间。

第 299、300 行：打印预训练所用时间数。

第 307~311 行：通过调用 `build_finetune_functions` 获取训练函数、验证模型（计算验证样本集上的分数）、测试模型（计算测试样本集上的分数），参数为：当前 MNIST 手写数字识别数据集、迷你批次中样本数、学习率。

第 317 行：定义 `patience`，如果连续 `patience` 次迭代代价函数值还没有明显改进，则设置 `done_looping` 为 `True`，终止训练过程。

第 320 行：如果发现验证样本集上分数有显著提高，增加 `patience` 的幅度。

第 323 行：验证样本集分数提高 0.5% 才视为有显著改善。

第 327 行：定义在验证样本集上验证分数是否有显著提高的频率。

第 329 行：定义记录所获得的最佳验证样本集上代价函数值的变量。

第 330 行：定义变量 `test_score` 为测试样本集上分数。

第 331 行：定义微调网络训练开始时间。

第 333 行：训练循环是否终止的标志，初始时为 `False`。

第 334 行：训练样本集遍历次数，初始时为 0。

第 336 行：训练样本集遍历次数 `epoch` 小于允许最大值，且循环停止标志 `done_looping` 为 `False` 时执行下面的循环。

第 337 行：将遍历次数加 1。

第 338 行：对所有迷你批次进行循环。

第 340 行：以一个迷你批次为参数，运行训练函数，计算代价函数值并更新网络参数。这里只需更新网络参数，所以不需要返回值。

第 341 行：计算总的迭代次数。

第 343 行：判断是否满足在验证样本集上进行验证的频率。

第 345 行：如果满足在验证样本集上进行验证的频率，先通过验证模型函数求出在验证样本集上的代价函数值。

第 346、347 行：求出验证样本集上代价函数在迷你批次上的平均值，即验证样本集上的分数。

第 348~354 行：打印验证样本集上的分数。

第 357 行：验证当前计算得到的验证样本集上的分数是否小于之前取得的分数最佳值。

第 360~362 行：如果当前计算得到的验证样本集上的分数小于之前取得的分数最佳值，判断是否有明显改进。如果有明显改进，则将 `patience` 值设置为迭代总次数乘以 `patience_increase`，即允许这么多次迭代验证样本集上代价函数值没有显著改进。

第 365 行：将本次求出的验证样本集上的分数赋给已取得的验证样本集上的分数。

第 366 行：将本次迭代记录为取得最佳效果的迭代。

第 369 行：调用测试模型函数 `test_model`，求出本迷你批次的代价函数值。

第 370 行：对这些值取平均值后得到测试样本集上的分数。

第 371~374 行：打印测试样本集上与分数相关的信息。

第 376~378 行：在一开始时，我们设置 `patience=5000`，表示如果迭代次数大于 5000 次时就会停止。训练开始后，假设在迭代次数为 1000 次时得到验证样本集上分数的最佳值，此时更新 `patience` 值为(5000, 2000)中的最大值，所以 `patience` 的值还是 5000。程序继续运行，当迭代次数为 4000 次时得到验证样本集上分数的最佳值，此时更新 `patience` 值为(5000, 8000)中的最大值，所以 `patience` 的值为 8000。如果此后一直没有出现验证样本集上分数的最佳值，则当迭代次数为 8000 次时训练循环就会结束。注：对一个迷你批次进行的处理，称为一次迭代。

第 380 行：定义微调网络训练结束时间。

第 381~387 行：打印训练的汇总时间。

下面来看隐藏层的定义，代码如下：

```
1 from __future__ import print_function
2 __docformat__ = 'restructuredtext en'
3 import os
4 import sys
5 import timeit
6 import numpy
7 import theano
8 import theano.tensor as T
9 from logistic_sgd import LogisticRegression, load_data
10
11 class HiddenLayer(object):
12     def __init__(self, rng, input, n_in, n_out, W=None, b=None,
13                  activation=T.tanh):
14         """
15         隐藏层主要用于多层感知器，也可以用于深度信念网络的
16         微调网络，默认时采用tanh激活函数，可以指定不同激活函数。
17         连接权值矩阵为（输入信号维度，本层神经元数量）。
18         参数
```

```

19 - rng : numpy随机数生成引擎
20 - input : 设计矩阵 (Design Matrix), 形状为 (迷你批次中样本数, 输入信号
21         维度)
22 - n_in : 输入信号维度
23 - n_out : 判断类别数, 输出层神经元数
24 - activation : 激活函数, 可以为Sigmoid函数或tanh函数等, 默认值为tanh函数
25 """
26 self.input = input
27
28 if W is None:
29     W_values = numpy.asarray(
30         rng.uniform(
31             low=-numpy.sqrt(6. / (n_in + n_out)),
32             high=numpy.sqrt(6. / (n_in + n_out)),
33             size=(n_in, n_out)
34         ),
35         dtype=theano.config.floatX
36     )
37     if activation == theano.tensor.nnet.sigmoid:
38         W_values *= 4
39
40     W = theano.shared(value=W_values, name='W', borrow=True)
41
42 if b is None:
43     b_values = numpy.zeros((n_out,), dtype=theano.config.floatX)
44     b = theano.shared(value=b_values, name='b', borrow=True)
45
46 self.W = W
47 self.b = b
48
49 lin_output = T.dot(input, self.W) + self.b
50 self.output = (
51     lin_output if activation is None
52     else activation(lin_output)
53 )
54 # parameters of the model
55 self.params = [self.W, self.b]

```

第 26 行: 定义属性 `input` 为本层的输入信号。

第 28~38 行: 如果连接权值矩阵没有指定, 则取 $\left[-\sqrt{\frac{6.0}{n_{in}+n_{out}}}, \sqrt{\frac{6.0}{n_{in}+n_{out}}}\right]$ 之间的随机数, 类型定义为 `floatX`, 这样可以使 Theano 在 GPU 上运行我们的代码。连接权值矩阵初始化需要考虑的因素很多, 包括对激活函数的选择。我们在默认条件下使用双曲正切函数 `tanh`, 如果使用 Sigmoid 函数, 根据 Sigmoid 函数与双曲正切函数的关系, 权值应该为默认值的 4 倍。其他激活函数没有相应的规则, 所以就默认与双曲正切函数相同, 并且将连接权值矩阵定义为 Theano 变量。

第 42~44 行: 如果没有定义偏移量, 则用全 0 初始化偏移量, 并且定义为 Theano 变量。

第 46 行: 定义属性 `W` 为连接权值矩阵。

第 47 行: 定义属性 `b` 为偏移量向量。

第 49 行: 定义线性输入和 `lin_output`。我们知道, `input` 为设计矩阵 (Design Matrix), 表示为 $\mathbb{R}^{n_1 \times m}$, 代表输入信号为 n_1 维, 共有 m 个样本, 第一层到第二层的连接权值矩阵为: $W^{(1,2)} \in \mathbb{R}^{n_2 \times n_1}$, 其中 n_2 为第二层即隐藏层神经元数量, 此时权值向量与输入设计矩阵的乘积 Wx 就是一个矩阵, 维度是 $\mathbb{R}^{n_2 \times m}$, 就是 m 个样本所对应的隐藏层状态。

第 50~53 行: 计算本层的输出值, 如果没有定义激活函数, 则直接返回线性输入和; 如果定义了激活函数, 则返回激活函数值。

下面来看进行实际模式分类的逻辑回归层定义, 代码如下:

```

1 from __future__ import print_function
2 __docformat__ = 'restructuredtext en'
3 import six.moves.cPickle as pickle
4 import gzip
5 import os
6 import sys
7 import timeit
8 import numpy
9 import theano
10 import theano.tensor as T
11
12 class LogisticRegression(object):
13     """
14     逻辑回归类处理多类别分类问题，也称为softmax回归
15     """
16
17     def __init__(self, input, n_in, n_out):
18         """
19         逻辑回归类构造函数
20         参数
21         - input: 输入信号，通常为一个迷你批次的样本
22         - n_in: 输入信号维度
23         - n_out: 分类类别数
24         """
25         # initialize with 0 the weights W as a matrix of shape (n_in, n_out)
26         self.W = theano.shared(
27             value=numpy.zeros(
28                 (n_in, n_out),
29                 dtype=theano.config.floatX
30             ),
31             name='W',
32             borrow=True
33         )
34
35         self.p_y_given_x = T.nnet.softmax(T.dot(input, self.W) + self.b)
36
37         # symbolic description of how to compute prediction as class whose
38         # probability is maximal
39         self.y_pred = T.argmax(self.p_y_given_x, axis=1)
40
41         # parameters of the model
42         self.params = [self.W, self.b]
43
44         # keep track of model input
45         self.input = input

```

第 17 行：定义逻辑回归类构造函数。

- ☐ input: 输入信号，通常为一个迷你批次的样本。
- ☐ n_in: 输入信号维度。
- ☐ n_out: 分类类别数。

第 26~33 行：定义属性 W 表示连接权值矩阵，用全 0 初始化连接权值矩阵，矩阵维度为 (n_in, n_out) ，并且定义其为 Theano 共享变量。

第 44 行：定义每个类别的出现概率，用 softmax 函数求第 i 个节点出现的概率的公式如下：

$$P_j = \frac{e^{x^{(i)}W_j + b_j}}{\sum_{c=1}^{n_out} e^{x^{(i)}W_c + b_c}}$$

式中， $x^{(i)}$ 表示第 i 个样本，其为 $\mathbb{R}^{1 \times n_in}$ 矩阵， W_j 为连接权值矩阵的第 j 列，其为 $\mathbb{R}^{n_in \times 1}$ ，此时 $x^{(i)}W_j$ 为两个向量点积，为一个数（标量）， b_j 为偏移量的第 j 个元素，上式分子部分为第 j 个神经元的激活值，分母部分为所有神经元的激活值之和，这个分数表示求第 j 类时

对应的概率。

在上式中，我们只考虑单一节点，而实际上传过来的 input 为 $\mathbb{R}^{m \times n_{in}}$ ，即每行代表一个样本， W 为 $\mathbb{R}^{n_{in} \times n_{out}}$ ，二者相乘之后得到 $\mathbb{R}^{m \times n_{out}}$ ，每行代表每个样本对应的逻辑回归层神经元上得到的值，再根据 numpy 的 broadcast 原理，将偏移量矩阵叠加到这个值上，就得到了逻辑回归神经元的线性输入和。假设用 $z_j^{(L)(i)}$ 表示逻辑回归层第 j 个神经元在最顶层 L 层的第 i 个样本所对应的线性输入和，则传递给 softmax 函数，其计算公式为：

$$p_{y_{given_x}} = \frac{e^{z_j}}{\sum_{c=1}^{n_{out}} e^{z_c}}$$

第 48 行：求出每个样本（每一行）概率最大的值，并将其作为输出。

第 51 行：定义属性 params 为本层参数，包括连接权值矩阵和偏移量。

第 54 行：定义属性 input 代表连接权值。

下面来看代价函数的定义，这里使用的是负对数似然函数，代码如下：

```
56 def negative_log_likelihood(self, y):
57     """
58     负对数似然函数通常作为网络的代价函数。这里使用的是平均值，而不
59     是和，因为平均值可以抵消迷你批次中样本数大小的影响
60     参数
61     - y: 迷你批次中正确分类结果
62     """
63     # y.shape[0] is (symbolically) the number of rows in y, i.e.,
64     # number of examples (call it n) in the minibatch
65     # T.arange(y.shape[0]) is a symbolic vector which will contain
66     # [0,1,2,... n-1] T.log(self.p_y_given_x) is a matrix of
67     # Log-Probabilities (call it LP) with one row per example and
68     # one column per class LP[T.arange(y.shape[0]),y] is a vector
69     # v containing [LP[0,y[0]], LP[1,y[1]], LP[2,y[2]], ...,
70     # LP[n-1,y[n-1]] and T.mean(LP[T.arange(y.shape[0]),y]) is
71     # the mean (across minibatch examples) of the elements in v,
72     # i.e., the mean log-likelihood across the minibatch.
73     return -T.mean(T.log(self.p_y_given_x)[T.arange(y.shape[0]), y])
```

代码中的注释比较难以理解，可以通过一个具体的实例来理解这段代码。以 MNIST 手写数字识别为例，假设迷你批次中样本数 $m=3$ ，则 $y=[6, 5, 8]$ ，因为我们只有三个样本，分别代表数字 6、5、8。而 $\text{self.p}_y\text{given}_x$ 为每个样本中每个类别出现的概率，其为 $\mathbb{R}^{m \times n_{out}}$ 矩阵，如下：

$$\begin{bmatrix} 0.01 & 0.01 & 0.01 & 0.01 & 0.01 & 0.01 & 0.91 & 0.01 & 0.01 & 0.01 \\ 0.02 & 0.02 & 0.02 & 0.02 & 0.02 & 0.82 & 0.02 & 0.02 & 0.02 & 0.02 \\ 0.03 & 0.03 & 0.03 & 0.03 & 0.03 & 0.03 & 0.03 & 0.03 & 0.73 & 0.03 \end{bmatrix}$$

将该矩阵进行 log 计算后得到以下矩阵：

$$\begin{bmatrix} -4.61 & -4.61 & -4.61 & -4.61 & -4.61 & -4.61 & -0.09 & -4.61 & -4.61 & -4.61 \\ -3.91 & -3.91 & -3.91 & -3.91 & -3.91 & -0.20 & -3.91 & -3.91 & -3.91 & -3.91 \\ -3.51 & -3.51 & -3.51 & -3.51 & -3.51 & -3.51 & -3.51 & -3.51 & -0.31 & -3.51 \end{bmatrix}$$

在此段代码中， $y.\text{shape}[0]=3$ 就是迷你批次大小， $T.\text{arange}(y.\text{shape}[0])$ 得到的是一个列表 $[0, 1, 2]$ ，其作为数组的第一维， $[T.\text{arange}(y.\text{shape}[0]), y]$ 形成一系列二维数组下标：

$$[(0,y[0]) \quad (1,y[1]) \quad (2,y[2])] = [(0,6) \quad (1,5) \quad (2,8)]$$

上式中的三个坐标分别对应于：

$$T.log(self.p_y_given_x)[0,6] = -0.09$$

$$T.log(self.p_y_given_x)[1,5] = -0.20$$

$$T.log(self.p_y_given_x)[2,8] = -0.31$$

将这三个数相加，取平均值后再加上负号就是最后的结果。

下面来看误差函数。在网络训练过程中，通常每隔一定的迭代次数，就需要在验证样本集上求出误差，看误差值是否有改进。如果误差值有改进则继续训练，如果误差值没有改进则停止训练，以提高模型的泛化能力，所以需要误差计算函数，代码如下：

```

75     def errors(self, y):
76         """
77         针对给定样本集，求出正确分类的样本数与总样本数的比值，以此来衡量模型
78         的分类效果
79         参数
80         - y: 正确的输出结果
81         """
82         # check if y has same dimension of y_pred
83         if y.ndim != self.y_pred.ndim:
84             raise TypeError(
85                 'y should have the same shape as self.y_pred',
86                 ('y', y.type, 'y_pred', self.y_pred.type)
87             )
88         # check if y is of the correct datatype
89         if y.dtype.startswith('int'):
90             # the T.neq operator returns a vector of 0s and 1s, where 1
91             # represents a mistake in prediction
92             return T.mean(T.neq(self.y_pred, y))
93         else:
94             raise NotImplementedError()

```

第 83~87 行：判断目标分类与预测的分类值的维度是否相同，不相同则报错。

第 89 行：只支持整数为分类结果，否则报错。

函数的具体计算过程以一个实例进行说明。

正确分类结果[8, 7, 9, 9, 8]代表 5 个样本，分别对应手写数字 8、7、9、9、8。

逻辑回归预测结果[8, 6, 2, 9, 8]代表对这 5 个样本的预测结果是手写数字 8、6、2、9、8。

我们看到，预测结果对了 3 个错了 2 个，所以误差应该为 0.4 (2/5)。

程序先求 $T.neq(y_pred, y)$ ，其为一个 `TensorVariable`，参数为计算出的结果向量数组和实际的结果向量数组，两个数组不相同结果为 1，相同时结果为 0，内容为[False, True, True, False, False]。然后调用 $T.mean(*)$ 就可以求出误差了。

下面来看 MNIST 手写数字识别数据集载入，代码如下：

```

1 from __future__ import print_function
2 __docformat__ = 'restructuredtext en'
3 import six.moves.cPickle as pickle
4 import gzip
5 import os
6 import sys
7 import timeit
8 import numpy
9 import theano
10 import theano.tensor as T
11
12 class MnistLoader(object):

```

```

13 def load_data(self, dataset):
14     data_dir, data_file = os.path.split(dataset)
15     if data_dir == "" and not os.path.isfile(dataset):
16         new_path = os.path.join(
17             os.path.split(__file__)[0],
18             "..",
19             "data",
20             dataset
21         )
22     if os.path.isfile(new_path) or data_file == 'mnist.pkl.gz':
23         dataset = new_path
24
25     if (not os.path.isfile(dataset)) and data_file == 'mnist.pkl.gz':
26         from six.moves import urllib
27         origin = (
28             'http://www.iro.umontreal.ca/~lisa/deep/data/'
29             'mnist/mnist.pkl.gz'
30         )
31         print('Downloading data from %s' % origin)
32         urllib.request.urlretrieve(origin, dataset)
33
34     print('... loading data')
35     # Load the dataset
36     with gzip.open(dataset, 'rb') as f:
37         try:
38             train_set, valid_set, test_set = pickle.load(f,
39                                                         encoding='latin1')
40         except:
41             train_set, valid_set, test_set = pickle.load(f)
42     def shared_dataset(data_xy, borrow=True):
43         data_x, data_y = data_xy
44         shared_x = theano.shared(numpy.asarray(data_x,
45                                                 dtype=theano.config.floatX),
46                                 borrow=borrow)
47         shared_y = theano.shared(numpy.asarray(data_y,
48                                                 dtype=theano.config.floatX),
49                                 borrow=borrow)
50         return shared_x, T.cast(shared_y, 'int32')
51
52     test_set_x, test_set_y = shared_dataset(test_set)
53     valid_set_x, valid_set_y = shared_dataset(valid_set)
54     train_set_x, train_set_y = shared_dataset(train_set)
55
56     rval = [(train_set_x, train_set_y), (valid_set_x, valid_set_y),
57            (test_set_x, test_set_y)]
58     return rval

```

第 13 行：定义数据载入接口，参数为 MNIST 手写数字识别数据集文件。

第 14~33 行：MNIST 手写数字识别数据集文件如果存在则打开该文件，如果不存在则从指定网址下载该文件。

第 52~54 行：将训练样本集、验证样本集、测试样本集定义为 Theano 的共享变量，以便在 Theano 预编译函数中使用。

第 56~58 行：以指定格式返回 MNIST 手写数字识别数据内容。

程序运行入口如下：

```

389 if __name__ == '__main__':
390     test_DBN()

```


第四部分 机器学习基础

- 生成式学习
- 支撑向量机

第 12 章

生成式学习

在本章中，将向大家介绍生成式学习，生成式网络在当前深度学习研究中属于非常热门的领域，受限玻尔兹曼机就是这种生成式网络。但是现在的生成式网络本身比较复杂，需要非常高深的数学模型来描述，普通人很难看懂，即使看懂了，也很难体会生成式学习在实践中到底是怎样被运用的。所以本章就从机器学习基础理论入手，向大家讲述什么是生成式学习。对这部分内容有了一定理解后，再回头看现在流行的生成式网络，相信大家就会有一种豁然开朗的感觉。

机器学习算法可以分为两大类：判决式学习、生成式学习。在判决式学习中，研究的通常是如 $p(y|x; \theta)$ 的模型，给定输入样本 x 及模型参数 θ ，求结果 y 的概率分布，即求以 x 为条件 y 的概率。以逻辑回归模型为例， $p(y|x; \theta)$ 可以表示为：

$$h_{\theta}(x) = g(\theta^T x + b)$$

式中， g 为 Sigmoid 函数，本式的计算结果就是 $y=1$ 时的概率。

但是生成式学习则是一类完全不同的学习方法。以模式分类问题为例，若我们拿到一批病理数据，共有两个维度 x_1 和 x_2 ，假设 $y=1$ 时为恶性肿瘤（用 \times ）， $y=0$ 时为良性肿瘤（用 \bullet 表示），如图 12.1 所示。

如果按照逻辑回归的算法来看，就是找到图中的直线，将良性肿瘤、恶性肿瘤区分开。如果新样本在直线的左上侧，则为恶性肿瘤，否则为良性肿瘤。因为我们在这里做的是在给定参数 θ 和 $x = [x_1, x_2]^T$ 的情况下，求 $y=1$ 和 $y=0$ 的概率，即 $p(y|x; \theta)$ ，因此称这类方法为判决式方法。

在生成式方法中，首先要做的是研究 $y=1$ 时所有恶性肿瘤样本的分布规律，建立恶性肿瘤模型；再研究 $y=0$ 时所有良性肿瘤样本的分布规律，建立良性肿瘤模型。当有一个新样本之后，看其是否与恶性肿瘤样本集模型相同，如果相同则为恶性肿瘤，如果与良性肿瘤样本集模型相同，则其为良性肿瘤。我们把这种方法称为生成式学习。

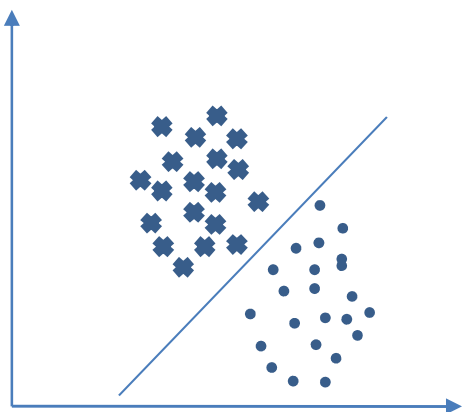


图 12.1 良性、恶性肿瘤的区分

我们知道所研究问题的类型后验概率分布 $p(y)$, $p(x|y=0)$ 为 $y=0$ 时样本的分布规律, $p(x|y=1)$ 为 $y=1$ 时样本的分布规律, 即 $p(x|y)$ 为以 y 为条件样本 x 的概率分布。知道这两个条件之后, 就可以根据贝叶斯定理求出以样本 x 为条件 y 的概率分布, 公式如下:

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)} \quad (1)$$

式中, 分母 $p(x)$ 的概率为在 $y=1$ 条件下 x 出现的概率加上在 $y=0$ 的条件下 x 出现的概率, 公式如下:

$$p(x) = p(x|y=1)p(y=1) + p(x|y=0)p(y=0)$$

这样式 (1) 的右侧就都是已知数了, 就可以求出 $p(y|x)$ 的概率了, 进而就可以判断新样本所属的类别了。

实际式 (1) 可以写成以下两个式子:

$$p(y=1|x) = \frac{p(x|y=1)p(y=1)}{p(x)}$$

$$p(y=0|x) = \frac{p(x|y=0)p(y=0)}{p(x)}$$

在实际判断的过程中, 哪个值大就判断新样本属于哪个类别。这两个公式的分母是一样的, 所以只计算分子的值就可以比较出它们的大小, 从而做出正确判断。

下面引入新的 argmax 算子, 并以一个实例为例来看这个算子的功能, 假设 $f(x)$ 函数定义如下:

$$f(x) = 5 - x^2$$

要求函数 $f(x)$ 取得最大值时 x 的值, 将其定义为 argmax 。函数 $f(x)$ 的计算很简单, 在 $x=0$ 的情况下取得最大值, 则可以表示为:

$$\operatorname{argmax}_x f(x) = 0$$

回到模式识别的例子，把 y 视为自变量， $p(y|\mathbf{x})$ 为 y 的函数，此时 y 的定义域为 $\{0,1\}$ ，即 y 只可以取 0 或 1，问题就可以描述为 y 取何值时 $p(y|\mathbf{x})$ 取得最大值，即可以表示为以下形式：

$$\operatorname{argmax}_y p(y|\mathbf{x}) = \operatorname{argmax}_y \frac{p(\mathbf{x}|y)p(y)}{p(\mathbf{x})} = \operatorname{argmax}_y p(\mathbf{x}|y)p(y)$$

在上式中，由于 $p(\mathbf{x})$ 是固定值，所以在比较两个数的最大值时，可以将其忽略不计，不影响最终结果。

12.1 高斯判别分析

下面来学习第一个生成式学习算法——高斯判别分析（GDA）。在这个模型中，认为在给定类别的条件下，样本集的分布 $p(\mathbf{x}|y)$ 是多变量正态分布。因此，在讲述高斯判别分析之前，先来研究多变量正态分布问题。

12.1.1 多变量高斯分布

在 n 维空间下，多变量正态分布也叫多变量高斯分布，其由参数均值向量 $\boldsymbol{\mu}$ 和协方差矩阵 $\boldsymbol{\Sigma}$ 组成。其中均值向量 $\boldsymbol{\mu} \in \mathbb{R}^n$ ，而协方差矩阵 $\boldsymbol{\Sigma} \in \mathbb{R}^{n \times n}$ ， $\boldsymbol{\Sigma}$ 0 且是对称阵，即为半定矩阵。多变量正态分布表示为 $N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ ，其概率密度可以表示为：

$$p(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{\frac{n}{2}} |\boldsymbol{\Sigma}|^{\frac{1}{2}}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})}$$

式中， $|\boldsymbol{\Sigma}|$ 为行列式。

对符合多变量高斯分布的随机变量 \mathbf{x} 的均值定义为：

$$\boldsymbol{\mu} = E[\mathbf{x}] = \int \mathbf{x} p(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) d\mathbf{x}$$

对任意实数 x 的协方差定义为：

$$\operatorname{Cov}(x) = E[(x - E(x))^2]$$

将其推广到随机向量则有：

$$\text{Cov}(\mathbf{x}) = \text{E}[(\mathbf{x} - \text{E}(\mathbf{x}))(\mathbf{x} - \text{E}(\mathbf{x}))^T]$$

也可以表示为:

$$\text{Cov}(\mathbf{x}) = \text{E}[\mathbf{x}\mathbf{x}^T] - \text{E}[\mathbf{x}](\text{E}[\mathbf{x}])^T$$

可以证明上面两个等式是等价的。

如果随机变量 \mathbf{x} 符合多变量正态分布, 即 $\mathbf{x} \sim N(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, 则有:

$$\boldsymbol{\Sigma} = \text{Cov}(\mathbf{x})$$

12.1.2 高斯判决分析公式

如果模式分类问题的特征 \mathbf{x} 是连续值随机变量, 可以用高斯判决分析来进行处理, 将给定类别 y 条件下样本集分布 $p(\mathbf{x}|y)$ 用多变量高斯分布来表示, 如下:

$$y \sim \text{Bernulli}(\Phi)$$

$$(\mathbf{x}|y = 0) \sim N(\boldsymbol{\mu}_0, \boldsymbol{\Sigma})$$

$$(\mathbf{x}|y = 1) \sim N(\boldsymbol{\mu}_1, \boldsymbol{\Sigma})$$

将其写为概率分布形式则有:

$$p(y) = \Phi^y (1 - \Phi)^{1-y}$$

$$p(\mathbf{x}|y = 0) = \frac{1}{(2\pi)^{\frac{n}{2}} |\boldsymbol{\Sigma}|^{\frac{1}{2}}} e^{-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_0)^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}_0)}$$

$$p(\mathbf{x}|y = 1) = \frac{1}{(2\pi)^{\frac{n}{2}} |\boldsymbol{\Sigma}|^{\frac{1}{2}}} e^{-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_1)^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}_1)}$$

在这里, 模型的参数为: Φ 、 $\boldsymbol{\mu}_0$ 、 $\boldsymbol{\mu}_1$ 、 $\boldsymbol{\Sigma}$ 。在不同类别下, 样本集的高斯分布中具有不同的均值 $\boldsymbol{\mu}_0$ 、 $\boldsymbol{\mu}_1$, 但是其协方差矩阵通常取相同的。

定义对数似然函数为:

$$\begin{aligned} \ell(\Phi, \boldsymbol{\mu}_0, \boldsymbol{\mu}_1, \boldsymbol{\Sigma}) &= \log \prod_{i=1}^m p(\mathbf{x}^{(i)}, y^{(i)}; \Phi, \boldsymbol{\mu}_0, \boldsymbol{\mu}_1, \boldsymbol{\Sigma}) \\ &= \log \prod_{i=1}^m p(\mathbf{x}^{(i)}|y^{(i)}; \Phi, \boldsymbol{\mu}_0, \boldsymbol{\mu}_1, \boldsymbol{\Sigma}) p(y^{(i)}; \Phi) \end{aligned}$$

如果想求对数似然函数的最大值, 则需要求出其对各参数的导数, 并令导数为 0, 则可以得到以下结果:

$$\Phi = \frac{1}{m} \sum_{i=1}^m 1\{y^{(i)} = 1\}$$

$$\mu_0 = \frac{\sum_{i=1}^m 1\{y^{(i)} = 0\} \mathbf{x}^{(i)}}{\sum_{i=1}^m 1\{y^{(i)} = 0\}}$$

$$\mu_1 = \frac{\sum_{i=1}^m 1\{y^{(i)} = 1\} \mathbf{x}^{(i)}}{\sum_{i=1}^m 1\{y^{(i)} = 1\}}$$

$$\Sigma = \frac{1}{m} \sum_{i=1}^m (\mathbf{x}^{(i)} - \mu_{y^{(i)}})(\mathbf{x}^{(i)} - \mu_{y^{(i)}})^T$$

这就是高斯判决分析的基本公式，下面我们来看一个具体的例子。

```

1 import matplotlib.pyplot as plt
2 from numpy import *
3
4 # 生成图12.1中右下侧样本
5 mean0=[2,3]
6 cov=mat([[1,0],[0,2]])
7 x0=random.multivariate_normal(mean0,cov,500).T
8 y0=zeros(shape(x0)[1])
9 # 生成图12.1中左上侧样本
10 mean1=[7,8]
11 n1=[7,8]
12 cov=mat([[1,0],[0,2]])
13 x1=random.multivariate_normal(mean1,cov,300).T
14 y1=ones(shape(x1)[1])
15
16 x=array([concatenate((x0[0],x1[0])),concatenate((x0[1],x1[1]))])
17 y=array([concatenate((y0,y1))])
18
19 m = x0.shape[1] + x1.shape[1]
20 phi = 1 / m *(x1.shape[1])
21 print('phi=%f' % phi)
22 mu0 = mean(x0, axis=1)
23 mu1 = mean(x1, axis=1)
24 print('mu0:\r\n%s' % mu0)
25 print('mu1:\r\n%s' % mu1)
26 x0 = x0.T
27 x1 = x1.T
28 x = x.T
29 x_mu = mat(concatenate([x0 - mu0, x1 - mu1]))
30 sigma = 1/m * (x_mu.T*x_mu)
31 print('sigma:\r\n%s' % sigma)

```

第 5~8 行：生成良性肿瘤样本集。首先在第 5 行定义均值向量为 `mean0`，然后定义协方差矩阵为 `cov`，再定义 `x0` 为符合这个多变量高斯分布的随机数数组，其形状为 (2, 500)，第一维是 `x0` 的 500 个值，第二维是 `x1` 的 500 个值。因为这些样本是良性肿瘤，所以 `y0` 为全 0 的数组，维度为 500。

第 10~14 行：生成恶性肿瘤样本集。首先在第 10 行定义均值向量为 `mean1`，然后定义协方差矩阵为 `cov`（与良性肿瘤样本集相同），再定义 `x1` 为符合这个多变量高斯分布的随机数数组，其形状为 (2, 300)，第一维是 `x0` 的 300 个值，第二维是 `x1` 的 300 个值。因为这些样本是恶性肿瘤，所以 `y1` 为全 1 的数组，维度为 300。

第 16 行：将恶性肿瘤和良性肿瘤 `x0` 元素加在一起作为 `x` 数组的第一维，将恶性肿瘤

和良性肿瘤 x_1 元素加在一起作为 x 数组的第二维。

第 17 行：将 y_0 和 y_1 的数据也合在一起，变为 1×800 的数组。

第 19 行：求出样本总数。

第 20 行：求出 Φ 值，公式如下：

$$\Phi = \frac{1}{m} \sum_{i=1}^m 1\{y^{(i)} = 1\}$$

第 22 行：求出值 μ_0 ，公式如下：

$$\mu_0 = \frac{\sum_{i=1}^m 1\{y^{(i)} = 0\} x^{(i)}}{\sum_{i=1}^m 1\{y^{(i)} = 0\}}$$

其实际上是对良性肿瘤样本集第一维所有样本值求平均值，再对第二维所有样本值求平均值，结果为二维向量。

第 23 行：求出值 μ_1 ，公式如下：

$$\mu_1 = \frac{\sum_{i=1}^m 1\{y^{(i)} = 1\} x^{(i)}}{\sum_{i=1}^m 1\{y^{(i)} = 1\}}$$

其实际上是对恶性肿瘤样本集第一维所有样本值求平均值，再对第二维所有样本值求平均值，结果为二维向量。

第 26 行：将 x_0 转置为 500×2 维，可以视为 500 个二维向量。

第 26 行：将 x_1 转置为 300×2 维，可以视为 300 个二维向量。

第 28 行：将 x 转置为 800×2 维，可以视为 800 个二维向量。

第 29 行：将新 x_0 的 500 个二维向量分别与均值 μ_0 相减，再将新 x_1 的 300 个二维向量分别与均值 μ_1 相减，然后将结果拼接在一起形成 800 个二维向量，并将其转换为 800×2 矩阵，其值代表式中 $x^{(i)} - \mu_{y^{(i)}}$ 。

第 30 行：按照公式求出 Σ 的值，因为其是 2×800 矩阵与 800×2 矩阵相乘，所以最后结果为 2×2 矩阵：

$$\Sigma = \frac{1}{m} \sum_{i=1}^m (x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - \mu_{y^{(i)}})^T$$

程序运行结果如下：

```
phi=0.375000
mu0:
[ 1.98318443  3.01042167]
mu1:
[ 7.01150658  7.95945908]
sigma:
[[ 1.04339532  0.03036238]
 [ 0.03036238  2.02042808]]
```

在给出一个新样本，如 $[9, 10]$ 时，可以按照以下方式进行区分：

$$\phi = \frac{1}{m} \sum_{i=1}^m 1\{y^{(i)} = 1\} = 0.375$$

$$\mu_0 = \frac{\sum_{i=1}^m 1\{y^{(i)} = 0\} \mathbf{x}^{(i)}}{\sum_{i=1}^m 1\{y^{(i)} = 0\}} = [1.983 \quad 3.010]$$

$$\mu_1 = \frac{\sum_{i=1}^m 1\{y^{(i)} = 1\} \mathbf{x}^{(i)}}{\sum_{i=1}^m 1\{y^{(i)} = 1\}} = [7.012 \quad 7.959]$$

$$\Sigma = \frac{1}{m} \sum_{i=1}^m (\mathbf{x}^{(i)} - \mu_{y^{(i)}})(\mathbf{x}^{(i)} - \mu_{y^{(i)}})^T = \begin{bmatrix} 1.043 & 0.030 \\ 0.030 & 2.020 \end{bmatrix}$$

代码如下：

```
1 import numpy as np
2
3 n = 2.0
4 pi = 3.1415926535
5 phi = 0.375
6 mu0 = np.asarray([1.983, 3.010])
7 mu1 = np.asarray([7.012, 7.959])
8 sigma = np.asarray([[1.043, 0.030], [0.030, 2.020]])
9
10 x = np.asarray([7.5, 8.9])
11
12 py1 = phi
13 py0 = 1 - phi
14 prefix_item = 1 / (np.power(2*pi, n/2)*np.power(np.linalg.det(sigma), 1/2.0))
15 exp_item = np.exp(-1/2.0*np.dot(np.dot((x-mu0), np.linalg.inv(sigma)), \
16 (x-mu0).T))
17 px_y0 = prefix_item * exp_item
18 print('px_y0=%.8e' % px_y0)
19 exp_item = np.exp(-1/2.0*np.dot(np.dot((x-mu1), np.linalg.inv(sigma)), \
20 (x-mu1).T))
21 px_y1 = prefix_item * exp_item
22 print('px_y1=%.8e' % px_y1)
23
24 px = px_y1 * py1 + px_y0 * py0
25 py1_x = px_y1*py1 / px
26 print('py1_x=%.8e' % py1_x)
27 py0_x = px_y0*py0 / px
28 print('py0_x=%.8e' % py0_x)
```

第 3 行：设定输入信号维度为 2。

第 4 行：设置圆周率常数。

第 5~8 行：设置多变量高斯分布参数，包括： $\Phi, \mu_0, \mu_1, \Sigma$ 。

第 10 行：假设新样本为[7.5, 8.9]，这是一个恶性肿瘤样本。

第 12 行：求出 $p(y=1) = \Phi^y(1-\Phi)^{1-y} = \Phi$ 。

第 13 行：求出 $p(y=0) = \Phi^y(1-\Phi)^{1-y} = 1-\Phi$ 。

第 14 行：求出 prefix_item 为下式时的值。

$$\text{prefix_item} = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}}$$

第 15 行：求出指数项 exp_item 在第二维时的值。

$$\text{exp_item} = e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_1)^T \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}_1)}$$

第 17 行：求出 $p(\mathbf{x}|y=1)$ 的值。

$$p(\mathbf{x}|y=1) = \frac{1}{(2\pi)^{\frac{n}{2}}|\boldsymbol{\Sigma}|^{\frac{1}{2}}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_1)^T \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}_1)} = \text{prefix_item} * \text{exp_item}$$

第 19 行：求出指数项 exp_item 在第一维时的值。

$$\text{exp_item} = e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_0)^T \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}_0)}$$

第 21 行：求出 $p(\mathbf{x}|y=0)$ 的值：

$$p(\mathbf{x}|y=0) = \frac{1}{(2\pi)^{\frac{n}{2}}|\boldsymbol{\Sigma}|^{\frac{1}{2}}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_0)^T \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}_0)} = \text{prefix_item} * \text{exp_item}$$

第 24 行：求出 $p(\mathbf{x})$ 。

$$p(\mathbf{x}) = p(\mathbf{x}|y=1)p(y=1) + p(\mathbf{x}|y=0)p(y=0)$$

第 25 行：求出 $p(y=1|\mathbf{x})$ ：

$$p(y=1|\mathbf{x}) = \frac{p(\mathbf{x}|y=1)p(y=1)}{p(\mathbf{x})}$$

第 27 行：求出 $p(y=0|\mathbf{x})$ ：

$$p(y=0|\mathbf{x}) = \frac{p(\mathbf{x}|y=0)p(y=0)}{p(\mathbf{x})}$$

运行结果如下：

```
px_y0=1.48111433e-11
px_y1=7.90871286e-02
py1_x=1.00000000e+00
py0_x=3.12127135e-10
```

由上面的运行结果可以看出， $p(\mathbf{x}|y=0)=1.48\text{e-}11$ ，而 $p(\mathbf{x}|y=1)=7.91\text{e-}02$ ，此时 $p(\mathbf{x}|y=1)>p(\mathbf{x}|y=0)$ ，所以可以判定这个新样本属于 $y=1$ 的类别，即该样本是恶性肿瘤。

我们还可以看这个样本的概率，首先恶性肿瘤的概率为 $p(y=1|\mathbf{x})=1\text{e}$ ，而良性肿瘤的概率为 $p(y=0|\mathbf{x})=3.12\text{e-}10$ ，几乎为零，这也说明上面的判断是正确的。

实际上，第一个例子是高斯判决分析的学习过程，而第二个例子则是高斯判决分析的运行过程。

为了加深对这一问题的理解，我们具体来实现一下高斯判决分析。我们已经向大家演示了怎么求出高斯判决分析参数 Φ 、 $\boldsymbol{\mu}_0$ 、 $\boldsymbol{\mu}_1$ 、 $\boldsymbol{\Sigma}$ ，以及怎样对新样本进行类别判定。下面将这些代码整合在一起，向大家演示高斯判决分析的具体实现。程序分为以下三部分。

- ❑ 训练代码：给定训练样本集，求出参数 Φ 、 $\boldsymbol{\mu}_0$ 、 $\boldsymbol{\mu}_1$ 、 $\boldsymbol{\Sigma}$ 。
- ❑ 运行：给定一个新样本，求出其所属类别。
- ❑ 学习：画出类别样本点、类别分界线、高斯分布等高线。

下面来看具体代码：

```

1 import numpy as np
2 import gaussian_nd as gaussian
3 import matplotlib.pyplot as plt
4
5 g_mu0 = [2.0, 3.0]
6 g_mu1 = [7.0, 8.0]
7 g_sigma = np.mat([[1.0, 0.0], [0.0, 2.0]])
8 c_type0 = 0
9 c_type1 = 1
10
11 def prepare_training_samples(type, mu, sigma, samples):
12     x = np.random.multivariate_normal(mu, sigma, samples).T
13     if c_type0 == type:
14         y = np.zeros(np.shape(x)[1])
15     elif c_type1 == type:
16         y = np.ones(np.shape(x)[1])
17     return x, y
18
19 def gda_nd(x, y):
20     m = sum([item.shape[1] for item in x])
21     phi = y[1].shape[0] / m
22     mu = [np.mean(item, axis=1) for item in x]
23     fx = [item.T - mu[idx] for idx, item in enumerate(x)]
24     rx = np.concatenate([item for item in fx])
25     rx = rx.T
26     sigma = (np.dot(rx, rx.T)) / m
27     return m, phi, mu, sigma
28
29 def get_bline(mu):
30     mid_point = [(mu[0][0] + mu[1][0])/2.0, (mu[0][1] + mu[1][1])/2.0]
31     k = (mu[1][1] - mu[0][1]) / (mu[1][0] - mu[0][0])
32     x = range(-2, 11)
33     y = [(-1.0/k)*(xi-mid_point[0]) + mid_point[1] for xi in x]
34     return x, y
35
36 def get_type_points(mu, sigma, type):
37     c_x = np.random.multivariate_normal(mu[type], sigma, 50).T
38     if (c_type0 == type):
39         c_y = np.zeros(c_x.shape[1])
40     else:
41         c_y = np.ones(c_x.shape[1])
42     return c_x, c_y
43
44 def prepare_contour_points(range0, range1, num):
45     x0 = np.linspace(range0[0], range0[1], num)
46     x1 = np.linspace(range1[0], range1[1], num)
47     return np.meshgrid(x0, x1)
48
49 def draw_type_points(c_x, mu, point_style):
50     plt.plot(c_x[0], c_x[1], point_style)
51     plt.plot(mu[0], mu[1], point_style, markersize=20)
52
53 def draw_bline(x, y):
54     plt.plot(x, y)
55
56 def draw_contour(x0, x1, z0):
57     cs = plt.contour(x0, x1, z0)
58     plt.clabel(cs, inline=1, fontsize=10)
59
60 def train():
61     x0, y0 = prepare_training_samples(c_type0, g_mu0, g_sigma, 5000)
62     x1, y1 = prepare_training_samples(c_type1, g_mu1, g_sigma, 3000)
63     rx = [x0, x1]
64     ry = [y0, y1]
65     return gda_nd(rx, ry)
66
67 def run(m, phi, mu, sigma):
68     x = np.array([8.5, 8.6])
69     n = x.shape[0]
70     py1 = phi
71     py0 = 1 - phi
72     prefix_item = 1 / (np.power(2*np.pi, 1/2.0)*np.power(np.linalg.det(\
73         sigma), 1/2.0))

```

```

74 exp_item0 = np.exp(-1/2.0*np.dot(np.dot((x-mu[0]), np.linalg.inv(\
75     sigma)), (x-mu[0]).T))
76 px_y0 = prefix_item * exp_item0
77 exp_item1 = np.exp(-1/2.0*np.dot(np.dot((x-mu[1]), np.linalg.inv(\
78     sigma)), (x-mu[1]).T))
79 px_y1 = prefix_item * exp_item1
80 print('0:%.8e; 1:%.8e' % (px_y0, px_y1))
81 if px_y0 > px_y1:
82     print('属于第0类')
83 else:
84     print('属于第1类')
85 px = px_y1 * py1 + px_y0 * py0
86 py0_x = px_y0 * py0 / px
87 py1_x = px_y1 * py1 / px
88 print('第0类概率: %.8e; 第1类概率: %.8e' % (py0_x, py1_x))
89
90 def demo(m, phi, mu, sigma):
91     plt.figure(1)
92     plt.clf()
93     blx, bly = get_bline(mu)
94     c0_x, c0_y = get_type_points(mu, sigma, 0)
95     c1_x, c1_y = get_type_points(mu, sigma, 1)
96     draw_type_points(c0_x, mu[0], 'xg')
97     draw_type_points(c1_x, mu[1], 'r')
98     draw_bline(blx, bly)
99     x0, x1 = prepare_contour_points([-2, 6], [-2, 7], 50)
100    z0 = gaussian.gaussian_2d(x0, x1, mu[0], sigma)
101    draw_contour(x0, x1, z0)
102    x0, x1 = prepare_contour_points([3, 11], [3, 12], 50)
103    z0 = gaussian.gaussian_2d(x0, x1, mu[1], sigma)
104    draw_contour(x0, x1, z0)
105    plt.show(1)
106
107 def startup():
108     m, phi, mu, sigma = train()
109     demo(m, phi, mu, sigma)
110     run(m, phi, mu, sigma)
111
112 if __name__ == '__main__':
113     startup()

```

第 112、113 行：程序的总入口点。

第 108 行：通过程序生成训练样本集数据，并通过高斯判决分析算法求出参数值：

Φ 、 μ_0 、 μ_1 、 Σ 。

第 109 行：根据类别 0、类别 1 的高斯分布，生成并画出数据点，并且画出二维高斯分布的等高线图，运行结果如图 12.2 所示。

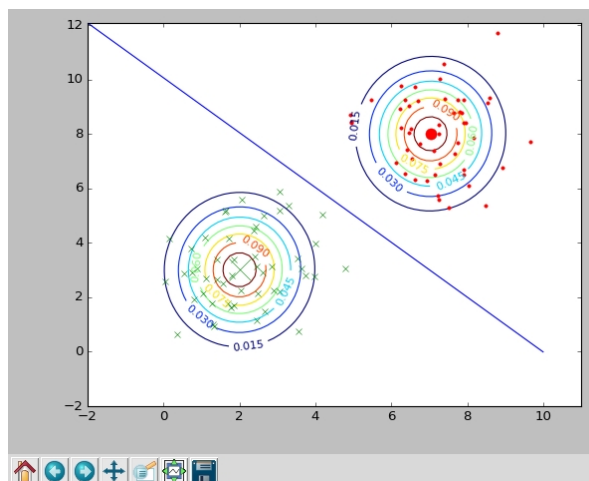


图 12.2 高斯判决分布图形示例

图中小叉子代表良性肿瘤样本点，小点代表恶性肿瘤样本点，同心圆分别代表良性肿瘤和恶性肿瘤高斯分布的等高线，斜直线代表所有 $p=0.5$ 的点组成的分割面。为了便于观察，我们使良性肿瘤生成了 50 个样本点，恶性肿瘤生成了 30 个样本点。

第 110 行：在已知参数 Φ 、 μ_0 、 μ_1 、 Σ 的情况下，判定 [7.5, 8.5] 点所属类别。为了验证正确性，还计算出了其分别属于类别 0 和类别 1 的概率。

下面分别来看这三个函数的实现，首先来看训练函数 `train`，该函数将从训练数据中求出参数 Φ 、 μ_0 、 μ_1 、 Σ 的值。

第 60 行：定义 `train` 函数。

第 61 行：生成类别 0 的训练数据，调用 `prepare_training_samples` 方法。参数：指定为类别 0，指定均值为全局变量，方差为全局变量，要生成 5000 个样本。

第 62 行：生成类别 1 的训练数据，调用 `prepare_training_samples` 方法。参数：指定为类别 1，指定均值为全局变量，方差为全局变量，要生成 3000 个样本。

第 63 行：将类别 0 和类别 1 的 `x` 数据组合在一起，`rx[0]` 为类别 0 的数据，`rx[1]` 为类别 1 的数据。

第 64 行：将类别 0 和类别 1 的 `y` 标签数据组合在一起，`ry[0]` 为类别 0 的标签，`ry[1]` 为类别 1 的标签。

第 65 行：调用 `gda_nd` 求出参数 Φ 、 μ_0 、 μ_1 、 Σ 的值。

再来看准备训练数据部分的程序。

第 5 行：定义全局变量 `g_mu0` 为类别 0 的均值 μ_0 ，由 `g_mu0` 可以看出，我们的例子是二维高斯判决分析。

第 6 行：定义全局变量 `g_mu1` 为类别 1 的均值 μ_1 。

第 7 行：定义全局变量 `g_sigma` 为协方差矩阵。

第 8 行：定义常量 `c_type0`，所有常量均以 `c_` 开头。

第 9 行：定义常量 `c_type1`。

第 11 行：定义 `prepare_training_samples` 函数。

❑ `type`：指定是类别 0 还是类别 1，取值为全局常量 `c_type0` 和 `c_type1`。

❑ `mu`：均值向量。

❑ `sigma`：协方差矩阵。

❑ `samples`：生成样本点数量。

第 12 行：调用 `multivariate_normal` 函数，生成多变量高斯分布数据点，其维度由均值向量维度决定，其形状为 (2, 样本数量)，即第一个元素为样本数量个 `x0` 的值，第二个元素为样本数量个 `x1` 的值，以 5000 个样本为例，其结构如下：

`[[x0_0, x0_1, x0_2, ..., x0_4999], [x1_0, x1_1, x1_2, ..., x1_4999]]`

样本点为 `[x0_0, x1_0], [x0_1, x1_1], [x0_2, x1_2], ..., [x0_4999, x1_4999]`

第 13、14 行：当类别为 0 时，`y` 标签值为 0。

第 15、16 行：当类别为 1 时，`y` 标签值为 1。

第 17 行：返回样本值和标签值。

下面来看高斯判决分析算法。

第 20 行：求出样本总数，因为 x 的第一个元素为类别 0 的样本值，第二个元素为类别 1 的样本值。先对 x 元素进行循环，求出每个元素的元素数并形成列表，在本例中即形成列表[5000,3000]，然后调用求和函数，求出总数为 8000。

第 21 行：求 ϕ 值，为类别 1 的样本数除以总的样本数。

第 22 行：求均值向量，对于 x 中的两个元素，分别对它们的第二维求均值。以类别 0 为 5000 样本，类别 1 为 3000 样为例，向大家演示运算过程。

```
[
#类别 0 样本
[ [x0_0, x0_1, x0_2, ..., x0_4999], [x1_0, x1_1, x1_2, ..., x1_4999]],
#类别 1 样本
[ [x0_0, x0_1, x0_2, ..., x0_2999], [x1_0, x1_1, x1_2, ..., x1_2999]]
]
```

结果为：

```
[ [x0_mean, x1_mean], [x0_mean, x1_mean] ]
```

第 23 行：先枚举 x 得到列表：

```
idx=0; item=[ [x0_0, x0_1, x0_2, ..., x0_4999], [x1_0, x1_1, x1_2, ..., x1_4999] ]
idx=1; item=[ [x0_0, x0_1, x0_2, ..., x0_2999], [x1_0, x1_1, x1_2, ..., x1_2999] ]
```

对这两个元素进行循环。

循环第一个元素时： $idx=0$ ， $item$ 的形状为 2×5000 ，我们先将其转置为 $item.T$ ，形状为 5000×2 ，然后再减去 $\mu[idx]=\mu[0]$ ，则对所有类别 0 的样本，我们就求出了 $x-\mu_0$ 的值，将其赋给 $fx[0]$ 。

循环到第二个元素时，用同样的方法求出 $x-\mu_1$ 的值，并将其赋给 $fx[1]$ 。

第 24 行：将 $fx[0]$ 、 $fx[1]$ 拼接为一个数组 rx ，此时 rx 的形状为 8000×2 ，也就是完整的 $x-\mu$ 的值。

第 25 行：对 rx 进行转置，变为 2×8000 ，以便之后进行绘图处理。

第 26 行：根据公式求出协方差矩阵 σ ：

$$\Sigma = \frac{1}{m} \sum_{i=1}^m (x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - \mu_{y^{(i)}})^T$$

第 27 行：返回所有高斯判决分析的参数。

以上就是高斯判决分析训练算法部分。下面我们来看高斯判决分析在图形上的表现形式，使大家对高斯判决分析有一个感性认识。

第 109 行：调用 `demo` 函数，将类别 0 和类别 1 的样本点、分界线、对应的高斯分布等高线图绘制到界面中。

第 90 行：定义 `demo` 函数。

- ❑ `m`: 样本总数。
- ❑ `phi`: 参数 Φ ，即类别 1 的概率。
- ❑ `mu`: 由均值`[mu0, mu1]`组成，其中 `mu0`、`mu1` 均为 `n` 维向量。
- ❑ `sigma`: 假设为 `n` 维高斯分布，`sigma` 为 `n`×`n` 矩阵。

第 91、92 行：初始化绘图系统。

第 93 行：调用 `get_bline` 函数，求出类别 0 和类别 1 的分隔线 `x` 点的坐标数组和 `y` 点的坐标数组。

第 94 行：调用 `get_type_points` 函数，根据多变量高斯分布参数，生成类别 0 的样本点集。

第 95 行：调用 `get_type_points` 函数，根据多变量高斯分布参数，生成类别 1 的样本点集。

第 96 行：调用 `draw_type_points` 函数，在界面中绘制类别 0 的样本点。

第 97 行：调用 `draw_type_points` 函数，在界面中绘制类别 1 的样本点。

第 98 行：调用 `draw_bline` 函数，绘制类别 0 和类别 1 的分隔线。

第 99 行：调用 `prepare_contour_points` 函数，获取类别 0 的 `x0` 值数组和 `x1` 值数组。

第 100 行：调用 `gaussian_2d` 函数，求出样本点`[x0, x1]`对应的高斯函数值，`z0` 的形状为 `n`×`n`。

第 101 行：调用 `draw_contour` 函数，绘制类别 0 的高斯分布等高线。

第 102 行：调用 `prepare_contour_points` 函数，获取类别 1 的 `x0` 值数组和 `x1` 值数组。

第 103 行：调用 `gaussian_2d` 函数，求出样本点`[x0, x1]`对应的高斯函数值，`z0` 的形状为 `n`×`n`。

第 104 行：调用 `draw_contour` 函数，绘制类别 1 的高斯分布等高线。

下面来看分割类别 0 和类别 1 的直线，代码解析如下。

第 30 行：均值 `mu0` 可以视为类别 0 样本点的中心，`mu1` 可以视为类别 1 样本点的中心，将这两个点连接起来，求出中点位置，并赋给 `mid_point`。

第 31 行：定义变量 `k` 为 `mu0` 点和 `mu1` 点直线的斜率 `k`。

第 32 行：定义分隔线 `x` 的取值范围。

第 33 行：先求出 `mu0` 到 `mu1` 直线的法线斜率，然后求出经过已知点和斜率的直线方程，循环 `x` 的值，求出法线上对应的 `y` 值。

第 34 行：返回分隔线 `x` 坐标点列表和 `y` 坐标点列表。

下面来看获取指定类别样本点函数，代码解析如下。

第 36 行：定义 `get_type_points` 函数。

❑ `mu` 为均值向量，第一个元素为类别 0 对应的均值向量，第二个元素为类别 1 对应的均值向量。

❑ `sigma`: 协方差矩阵。

❑ `type`: 常量 `c_type0` 和 `c_type1`，分别代表求类别 0 和类别 1。

第 37 行：调用 `np.random.multivariate` 函数，参数为对应类别的均值向量、协方差矩阵、要生成的样本数。

第 38、39 行：如果是类别 0，则生成全 0 数组。

第 40、41 行：如果是类别 1，则生成全 1 数组。

第 42 行：返回样本值集和样本标签集。

下面来看将指定类别样本点绘制到界面图形中，代码解析如下。

第 49 行：定义 `draw_type_points` 函数。

❑ `c_x`：样本值数组，形状为 $2 \times$ 样本数量。

❑ `mu`：均值向量。

❑ `point_style`：绘制点的风格，包括点的形状和颜色。

第 50 行：绘制样本点。

第 51 行：在样本点中心绘制一个标志。

下面来看高斯分布数据点的生成函数，代码解析如下。

第 44 行：定义 `prepare_contour_points` 函数。

❑ `range0`：x0 坐标的最小值与最大值。

❑ `range1`：x1 坐标的最小值与最大值。

❑ `num`：生成多少个样本点。

第 45 行：生成 x0 的坐标值列表，在最小值与最大值之间等距取 `num` 个点。

第 46 行：生成 x1 的坐标值列表，在最小值与最大值之间等距取 `num` 个点。

第 47 行：将 x0 的坐标值列表、x1 的坐标值列表两两组合，形成新样本列表，并返回该列表。

程序中用到的 `np.meshgrid` 函数，可以通过下面这个小例子来理解：

```
1 import numpy as np
2
3 nx, ny = (5, 3)
4 x = np.linspace(0, 1, nx)
5 y = np.linspace(0, 1, ny)
6 xv, yv = np.meshgrid(x, y)
7 print('xv:%s' % xv)
8 print('yv:%s' % yv)
```

运行结果为：

```
xv:[[ 0.    0.25  0.5   0.75  1. ]
 [ 0.    0.25  0.5   0.75  1. ]
 [ 0.    0.25  0.5   0.75  1. ]]
yv:[[ 0.    0.    0.    0.    0. ]
 [ 0.5  0.5  0.5  0.5  0.5]
 [ 1.    1.    1.    1.    1. ]]
```

如上所示，x 为 $[0, 0.25, 0.5, 0.75, 1.0]$ ，y 为 $[0, 0.5, 1.0]$ 。xv 是二维数组，将 x 自身重复 y 数组的长度。yv 也是二维数组，第一维长度也为 y 数组的长度，每个元素的值为 y 数组对应位置的值，但是重复 x 数组长度的次数。

下面来看二维高斯分布求值，代码在 `gaussian` 模块中，如下：

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from matplotlib import cm
4 from mpl_toolkits.mplot3d import Axes3D
5
6 # 声明全局变量，表示二维高斯分布
7 g_mu = np.array([0., 1.])
8 g_sigma = np.array([[ 1., -0.5], [-0.5, 1.5]])
9
```

```

10 def multivariate_gaussian(pos, mu, sigma):
11     """
12     多维高斯函数实现，由gaussian_xd来调用
13     """
14     n = mu.shape[0]
15     Sigma_det = np.linalg.det(sigma)
16     Sigma_inv = np.linalg.inv(sigma)
17     N = np.sqrt((2*np.pi)**n * Sigma_det)
18     # This einsum call calculates (x-mu)T.Sigma-1.(x-mu) in a vectorized
19     # way across all the input variables.
20     fac = np.einsum('...k,kl,...l->...', pos-mu, Sigma_inv, pos-mu)
21     return np.exp(-fac / 2) / N
22
23 def gaussian_2d(x0, x1, mu, sigma):
24     """
25     二维高斯函数调用接口，重点工作是将x0和x1变换成多维高斯函数需要的
26     格式
27     """
28     pos = np.empty(x0.shape + (2,))
29     pos[:, :, 0] = x0
30     pos[:, :, 1] = x1
31     return multivariate_gaussian(pos, mu, sigma)

```

第 23 行：定义 gaussian_2d 函数，这是调用接口。

- ☐ x0: 调用 np.meshgrid 后所得的第一个元素。
- ☐ x1: 调用 np.meshgrid 后所得的第二个元素。
- ☐ mu: 均值向量。
- ☐ sigma: 协方差矩阵。

第 28 行：定义 pos 三维数组，形状为 x0 前两维的形状，第三维长度为 2。

第 29 行：定义 pos 数组第三维第一个元素为 x0 的元素。

第 30 行：定义 pos 数组第三维第二个元素为 x1 的元素。

第 31 行：调用 multivariate_gaussian 函数，计算高斯分布的值。

这段代码是比较难以理解的，还是以上面的例子为例，x 为[0, 0.25, 0.5, 0.75, 1.0]，而 y 为[0, 0.5, 1.0]，按照下面的示例程序：

```

1 import numpy as np
2
3 nx, ny = (5, 3)
4 x = np.linspace(0, 1, nx)
5 y = np.linspace(0, 1, ny)
6 x0, x1 = np.meshgrid(x, y)
7 pos = np.empty(x0.shape + (2,))
8 pos[:, :, 0] = x0
9 pos[:, :, 1] = x1
10 print(pos)

```

运行结果如下：

```

[[[ 0.  0. ]
  [ 0.25 0. ]
  [ 0.5  0. ]
  [ 0.75 0. ]
  [ 1.  0. ]]]

[[[ 0.  0.5 ]
  [ 0.25 0.5 ]
  [ 0.5  0.5 ]
  [ 0.75 0.5 ]
  [ 1.  0.5 ]]]

[[[ 0.  1. ]
  [ 0.25 1. ]
  [ 0.5  1. ]
  [ 0.75 1. ]
  [ 1.  1. ]]]]

```

下面来看高斯分布的具体计算函数 `multivariate_gaussian`，代码解析如下。

第 10 行：定义高斯分布计算函数 `multivariate_gaussian`。

❑ `pos`：所有数据点坐标列表。

❑ `mu`：均值向量。

❑ `sigma`：协方差矩阵。

第 14 行：求出高斯分布的维度

第 15 行：求出协方差 `sigma` 的 `det` 值（会在后面的计算中用到）。

第 16 行：求出协方差 `sigma` 的逆（会在后面的指数计算中用到）。

第 17~21 行：计算高斯分布的值并返回，计算公式如下：

$$p(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{\frac{n}{2}} |\boldsymbol{\Sigma}|^{\frac{1}{2}}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})}$$

程序中采用分步计算的方式，仔细跟踪就可以对应到公式中的各项，这里就不再逐一讲解其中的对应关系了。

下面来看二维高斯分布等高线绘制程序，代码解析如下。

第 56 行：定义 `draw_contour` 函数。

❑ `x0`：x0 坐标值点。

❑ `x1`：x1 坐标值点。

❑ `z0`：(x0, x1) 点所对应的高斯分布值。

第 57 行：调用 `pyplot.contour` 函数绘制等高线。

第 58 行：在等高线上标注其所对应的数值。

至此，我们就将高斯判决分析图形展示部分的代码讲解完了，下面我们来看模型在正式运行条件下，怎样区别一个新样本属于哪个类别，那就是使用 `run` 函数，代码解析如下。

第 67 行：定义 `run` 函数。

❑ `m`：样本总数。

❑ `phi`：类别 1 的概率 Φ 。

❑ `mu`：均值矩阵，包括类别 0 的均值矩阵 `mu[0]` 和类别 1 的均值矩阵 `mu[1]`。

❑ `sigma`：协方差矩阵。

第 68 行：定义新的待分类的样本 `x`，根据经验，该点应该是类别 1，对应的是恶性肿瘤。

第 69 行：求出这个问题的维度，这里是二维。

第 70 行：求出类别 1 的概率 `py1`，其值是 Φ 。

第 71 行：求出类别 0 的概率 `py0`，其值为 $1-\Phi$ 。

第 72~76 行：根据公式求出 $p(\mathbf{x}|\mathbf{y}=0)$ 的值。

$$p(\mathbf{x}|\mathbf{y}=0) = \frac{1}{(2\pi)^{\frac{n}{2}} |\boldsymbol{\Sigma}|^{\frac{1}{2}}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_0)^T \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}_0)}$$

第 77~79 行：根据公式求出 $p(\mathbf{x}|y=1)$ 的值。

$$p(\mathbf{x}|y=1) = \frac{1}{(2\pi)^{\frac{n}{2}} |\Sigma|^{\frac{1}{2}}} e^{-\frac{1}{2}(\mathbf{x}-\mu_1)^T \Sigma^{-1}(\mathbf{x}-\mu_1)}$$

第 81、82 行：如果 $p(\mathbf{x}|y=0) > p(\mathbf{x}|y=1)$ ，则属于第 0 类。

第 83、84 行：如果 $p(\mathbf{x}|y=0) < p(\mathbf{x}|y=1)$ ，则属于第 1 类。在本示例中，新样本属于第 1 类。到这里就已经求出新样本类别了，在实际应用中就可以结束了。但是为了给大家一个直观的认识，继续求出新样本属于类别 0 的概率和类别 1 的概率。

第 85 行：求出 $p(\mathbf{x}) = p(\mathbf{x}|y=1)p(y=1) + p(\mathbf{x}|y=0)p(y=0)$ 。

第 86 行：根据公式求出 $p(y=0|\mathbf{x})$ 。

$$p(y=0|\mathbf{x}) = \frac{p(\mathbf{x}|y=0)p(y=0)}{p(\mathbf{x})}$$

第 87 行：根据公式求出 $p(y=1|\mathbf{x})$ 。

$$p(y=1|\mathbf{x}) = \frac{p(\mathbf{x}|y=1)p(y=1)}{p(\mathbf{x})}$$

输出结果为：

```
0:7.45521891e-14; 1:8.57182333e-02
属于第1类
第0类概率: 1.44955914e-12; 第1类概率: 1.00000000e+00
```

由此可以看出， $p(\mathbf{x}|y=1)$ 远远大于 $p(\mathbf{x}|y=0)$ ，所以可以判定其属于第 1 类。从概率上看，其属于第 0 类的概率基本为零，而属于第 1 类的概率基本为 1，属于相当有把握的判断。

在高斯判决分析中，特征向量 \mathbf{x} 是连续型实数值向量。但是如果特征向量是离散值时怎么处理呢？下面我们就来讲一下朴素贝叶斯模型，这种模型是有别于高斯判决分析的另一种生成式学习算法。

12.2 朴素贝叶斯

12.2.1 朴素贝叶斯分类器

以垃圾邮件分类为例，我们想采用机器学习算法来过滤垃圾邮件，此时需要区分邮件是未经许可发送的商业邮件还是普通邮件。这就要求算法可以自动过滤邮件，区分垃圾邮件和正常邮件，并把它们放入不同的文件夹中。这就是 Outlook、QQ 邮箱等所做的工作，其实就是文本分类的一个特例。

我们的训练样本集是一批经过人工标记的邮件，包括普通邮件和垃圾邮件，我们的任务就是通过学习这些标记的邮件，掌握邮件分类的方法。

首先，对所有邮件进行遍历，找出所有不同的单词，组成一个单词表，如下：

[a, aardvark, aardwolf, ..., buy, ..., zygmurgy]

然后，对于每一封邮件，如果包含某个单词，就在该单词相应位置标 1；如果没有出现这个单词则标为 0，例如一封邮件只包含 buy 和 aardwolf 两个单词，则这封邮件的特征向量为：

[0, 0, 1, ..., 1, ..., 0]

由此可见，在我们的模型中，每封邮件特征向量的维度为单词表的长度，特征向量中的每个元素的取值为 0、1，表示是否包含单词表对应位置的单词。如果单词表长度为 22000，邮件的特征向量的维度就是 22000 维，此时其维度是相当高的。

但是有很多单词，比如 a、the、of 等，是否包含它们是不影响我们对垃圾邮件的分类的，所以为了降低特征向量的维度，可以将这些不影响垃圾邮件分类的单词从单词表中除去，包括通常在垃圾邮件中出现的单词即可。这样，假设单词表最终长度为 3000，则特征向量的维度也是 3000。

我们知道，生成式学习要研究的是 $p(x|y)$ ，则可以把这个问题表示为：

$$p(x_1, x_2, \dots, x_{2999}, x_{3000} | y) = p(x_1 | y) p(x_2 | y, x_1) p(x_3 | y, x_1, x_2) \cdots p(x_{3000} | x_1, x_2, \dots, x_{2999})$$

但是如果直接按照上式来求解，对上式中各项条件概率求解非常复杂，需要找到一种简便的计算方法，这就是朴素贝叶斯假设。

以垃圾邮件分类问题为例，朴素贝叶斯假设认为，假定邮件是垃圾邮件，每个单词出现的概率是互相独立的，即假设两个单词所对应的特征向量元素为 x_i 和 x_j ，则有如下关系：

$$p(x_i | y) = p(x_i | y, x_j)$$

即当邮件是垃圾邮件时，出现单词 i 的概率与是否出现单词 j 无关。以一个具体的实例为例，假设 $i=299$ 对应的单词为 buy，而 $j=1899$ 对应的单词为 price，朴素贝叶斯假设认为，如果邮件为垃圾邮件，单词 buy 出现的概率为 p_1 ，那么无论单词 price 是否出现，单词 buy 出现的概率都不会有任何变化，可以表示为：

$$p(x_{299} | y) = p(x_{299} | y, x_{1899})$$

注意：在这里并不是说单词 buy 和 price 没有关系，因为出现单词 buy 的时候，非常有可能出现单词 price。在这里只是说，当邮件是垃圾邮件时，单词 buy 出现的概率 p_1 ，与是否出现单词 price 无关。若是单词 buy 与 price 无关，则应该表示为：

$$p(x_{\text{buy}}) = p(x_{\text{buy}} | x_{\text{price}})$$

而这显然是不正确的。

再回到原来的问题，根据朴素贝叶斯假设，可以得到朴素贝叶斯分类器，即：

$$\begin{aligned} p(x_1, x_2, \dots, x_{2999} | y) &= p(x_1 | y) p(x_2 | y, x_1) p(x_3 | y, x_1, x_2) \cdots p(x_{3000} | x_1, x_2, \dots, x_{2999}) \\ &= p(x_1 | y) p(x_2 | y) p(x_3 | y) \cdots p(x_{2999} | y) p(x_{3000} | y) \end{aligned}$$

$$= \prod_{i=1}^n p(x_i|y)$$

当为垃圾邮件（ $y=1$ ）时，单词 i 出现的概率如下：

$$\Phi_{i|y=1} = p(x_i = 1|y = 1)$$

当不为垃圾邮件（ $y=0$ ）时，单词 i 出现的概率如下：

$$\Phi_{i|y=0} = p(x_i = 1|y = 0)$$

邮件为垃圾邮件的概率为：

$$\Phi_y = p(y = 1)$$

设训练样本集为： $\{(x^{(i)}, y^{(i)}); i = 1, 2, \dots, m\}$

可以将联合似然函数写为：

$$\mathcal{L}(\Phi_y, \Phi_{j|y=0}, \Phi_{j|y=1}) = \prod_{i=1}^m p(x^{(i)}, y^{(i)})$$

以 $\Phi_y, \Phi_{j|y=0}, \Phi_{j|y=1}$ 为参数，可以得到最大似然估计为：

$$\Phi_{j|y=1} = \frac{\sum_{i=1}^m 1\{x_j^{(i)} = 1 \text{ and } y^{(i)} = 1\}}{\sum_{i=1}^m 1\{y^{(i)} = 1\}}$$

当邮件为垃圾邮件时，单词 j 出现的概率为出现单词 j 且为垃圾邮件的样本数除以垃圾邮件样本数。

当邮件不是垃圾邮件（ $y=0$ ）时，单词 j 出现的概率为：

$$\Phi_{j|y=0} = \frac{\sum_{i=1}^m 1\{x_j^{(i)} = 1 \text{ and } y^{(i)} = 0\}}{\sum_{i=1}^m 1\{y^{(i)} = 0\}}$$

当邮件不是垃圾邮件时，单词 j 出现的概率为不是垃圾邮件且出现单词 j 的样本数除以不是垃圾邮件的样本数。

垃圾邮件出现的概率为：

$$\Phi_y = \frac{\sum_{i=1}^m 1\{y^{(i)} = 1\}}{m}$$

当需要对一个新样本进行预测时，可以根据以下公式进行预测：

$$\begin{aligned} p(y = 1|x) &= \frac{p(x|y = 1)p(y = 1)}{p(x)} \\ &= \frac{(\prod_{i=1}^n p(x_i|y = 1))p(y = 1)}{(\prod_{i=1}^n p(x_i|y = 1))p(y = 1) + (\prod_{i=1}^n p(x_i|y = 0))p(y = 0)} \end{aligned}$$

$$\begin{aligned}
 p(y=0|\mathbf{x}) &= \frac{p(\mathbf{x}|y=0)p(y=0)}{p(\mathbf{x})} \\
 &= \frac{(\prod_{i=1}^n p(x_i|y=0))p(y=0)}{(\prod_{i=1}^n p(x_i|y=1))p(y=1) + (\prod_{i=1}^n p(x_i|y=0))p(y=0)}
 \end{aligned}$$

可以通过比较 $p(y=1|\mathbf{x})$ 和 $p(y=0|\mathbf{x})$ 值的大小，确定其是垃圾邮件还是普通邮件。

12.2.2 拉普拉斯平滑

到目前为止，我们所讲的朴素贝叶斯模型，对于处理垃圾邮件分类等问题是非常有效的。但是这种方法有一个潜在的问题，就是要求训练样本集足够完善，否则就会有问题。

比如我们给某个公司做了一个垃圾邮件分类系统，这个公司是一个传统行业公司，主要业务是机械加工，因此其单词表中的词汇均与机械行业有关。但是该公司目前想引入深度学习技术，所以邮件中可能出现“深度学习”这个词，那如果用朴素贝叶斯模型算法来处理这种情况，会出现什么结果呢？

假设发现“深度学习”这个单词之后，我们先将其加入单词表的末尾，序号为 3001，这时需要更新贝叶斯模型的参数：

$$\phi_{3001|y=1} = \frac{\sum_{i=1}^m 1\{x_{3001}^{(i)} = 1 \text{ and } y^{(i)} = 1\}}{\sum_{i=1}^m 1\{y^{(i)} = 1\}} = 0$$

因为训练样本集中不包含“深度学习”这个单词且为垃圾邮件的样本，则：

$$\phi_{3001|y=0} = \frac{\sum_{i=1}^m 1\{x_{3001}^{(i)} = 1 \text{ and } y^{(i)} = 0\}}{\sum_{i=1}^m 1\{y^{(i)} = 0\}} = 0$$

同理，因为训练样本集中不包含“深度学习”这个单词且不是垃圾邮件的样本。

下面来求其不是垃圾邮件的概率：

$$p(y=0|\mathbf{x}) = \frac{(\prod_{i=1}^n p(x_i|y=0))p(y=0)}{(\prod_{i=1}^n p(x_i|y=1))p(y=1) + (\prod_{i=1}^n p(x_i|y=0))p(y=0)} = \frac{0}{0}$$

式中， $n=3001$ ，当 $i=3001$ 时，所对应的项为 0，因为上式中每项均为连乘，任一项为 0 则导致整个项为 0，所以上式结果为 0 除以 0，没有定义。

求该邮件是否为垃圾邮件的概率：

$$p(y=1|\mathbf{x}) = \frac{(\prod_{i=1}^n p(x_i|y=1))p(y=1)}{(\prod_{i=1}^n p(x_i|y=1))p(y=1) + (\prod_{i=1}^n p(x_i|y=0))p(y=0)} = \frac{0}{0}$$

同理，其也是 0 除以 0，没有定义。

这说明在这种情况下应用朴素贝叶斯算法是存在问题的。那么怎么来解决这个问题

呢？这就需要用到下面我们要讲到的拉普拉斯平滑方法。

据说这一方法是拉普拉斯在解决计算明天太阳升起的概率时发明的。我们知道，太阳已经升起 5000 年了，即有 5000×365 天，那么明天太阳不升起的概率就是 0 吗？这显然是不正确的。但是当时计算概率的公式为：

$$p = \frac{\text{事件发生数}}{\text{事件发生数} + \text{事件未发生数}}$$

这里事件发生数为 0，则确定概率为 0。这显然不是特别合理，我们不能因为没有看到事件发生，就认为该事件不可能发生。

在这个例子里，你可能认为明天太阳不升起的概率为 0 是合理的。那我们再看下面这个例子，某县中学举行足球联赛，假设下面为二中与其他中学的比赛结果，请预测二中与十中的比赛结果。

	一中	三中	四中	五中	六中	七中	八中	九中	十中
	负	负	负	负	负	负	负	负	?

在这里我们就不能因为其连输 8 场，就断定其第 9 场一定会赢，虽然输的概率会大一些。

再回到太阳升起的问题，拉普拉斯假设，未来两天我们将分别看到太阳升起和太阳不升起，则明天太阳不升起的概率为：

$$p = \frac{\text{事件发生数} + 1}{(\text{事件发生数} + 1) + (\text{事件未发生数} + 1)} = \frac{0 + 1}{(0 + 1) + (5000 \times 365 + 1)}$$

这样明天太阳不升起的概率就不为 0 了，而是一个非常小的值。

对于更贴近实际生活的中学足球联赛的例子来说：

$$p = \frac{\text{事件发生数} + 1}{(\text{事件发生数} + 1) + (\text{事件未发生数} + 1)} = \frac{0 + 1}{(0 + 1) + (8 + 1)} = 0.1 = 10\%$$

所以预测 2 中与 10 中比赛取胜的概率为 10%。

以上就是拉普拉斯平滑的实例，下面用数学术语来描述拉普拉斯平滑。

假设我们想把文章划分为语文、数学、物理、化学、生物、历史、地理、政治、英语等类别，用变量 z 来表示各类别，则 $z \in \{1, 2, \dots, k\}$ ，1 代表语文，2 代表数学，以此类推。假设其为某一类别的概率为：

$$\phi_j = \frac{\sum_{i=1}^m 1\{z^{(i)} = j\}}{m}$$

根据拉普拉斯平滑，我们将得到以下模型：

$$\phi_j = \frac{\sum_{i=1}^m 1\{z^{(i)} = j\} + 1}{m + k}$$

在这里将每个事件的样本数均加上 1，所以分子加 1，分母加 k 。

回到朴素贝叶斯模型，如下：

$$\phi_{j|y=1} = \frac{\sum_{i=1}^m 1\{x_j^{(i)} = 1 \text{ and } y^{(i)} = 1\} + 1}{\sum_{i=1}^m 1\{y^{(i)} = 1\} + 2}$$

即向训练样本集中加入了两个样本，一个是邮件为垃圾邮件且该单词出现，另一个是邮件为垃圾邮件但该单词未出现。

$$\phi_{j|y=0} = \frac{\sum_{i=1}^m 1\{x_j^{(i)} = 1 \text{ and } y^{(i)} = 0\} + 1}{\sum_{i=1}^m 1\{y^{(i)} = 0\} + 2}$$

即向训练样本集中加入了两个样本，一个是邮件不为垃圾邮件但该单词出现，另一个是邮件不为垃圾邮件但该单词未出现。

对于邮件是垃圾邮件的概率，因为训练样本集中肯定会有大量垃圾邮件和非垃圾邮件，这个概率肯定不会为 0，因此就没有必要应用拉普拉斯平滑技术了。

在垃圾邮件分类的例子中，关于特征向量，我们只定义了某个单词是否出现，用 0、1 表示，现在稍微改变一下，记录某个单词在邮件中出现的次数，并且规定最大次数为 100 次，则特征向量中每个元素取值为{0, 1, ..., 100}，这时如果应用朴素贝叶斯算法，公式就变为以下形式：

$$\phi_{j=c|y=1} = \frac{\sum_{i=1}^m 1\{x_j^{(i)} = c \text{ and } y^{(i)} = 1\} + 1}{\sum_{i=1}^m 1\{y^{(i)} = 1\} + 101}$$

上式中 c 的取值为 0 至 100 间的数，分母中加 101 是因为 j 的所有取值情况共有 0~100 种可能性，即 101 种可能性。

12.2.3 多项式事件模型

在文本分类领域，朴素贝叶斯模型被称为多变量伯努利事件模型。这种模型认为，邮件发送者先根据 $p(y)$ 的概率决定要发的邮件是正常邮件还是垃圾邮件。假设他决定发送垃圾邮件，就会根据计算出的概率 $p(x_i = 1|y = 1) = \phi_{i|y=1}$ 确定是否包含某个单词，则整篇邮件的概率为 $(\prod_{i=1}^n p(x_i|y))p(y)$ 。

对于垃圾邮件分类问题，还可以使用多项式事件模型。在这个模型中，特征向量第 i 个位置元素的值为该位置的单词在单词表中的索引号，假设单词表有 3000 个单词，用 V 来表示，而且他要发一封垃圾邮件，单词表如下：

1, 2, 299, ..., 366, ..., 1803, ..., 2531, ...
[aba, abc, ..., book, ..., buy, ..., please, ..., this, ...]

第一行是单词表单词所对应的序号，假设最终我们的邮件为：

please buy this book

那么其所对应的特征向量为：

[1803, 366, 2531, 299]

由此可以看出，每个邮件对应的特征向量的维度是不同的，是邮件的长度。

在多项式事件模型中，邮件发送者根据概率 $p(y)$ 确定发送垃圾邮件还是普通邮件，从第一个单词开始，每个位置都以多项式分布 $p(x_i|y=1)$ ，选择需要出现的单词。各个单词除符合一样的多项式分布外，彼此间相互独立。我们一共选择 n_j 个单词， j 代表为第 j 封邮件。这封邮件的概率同样可以表示为 $(\prod_{i=1}^n p(x_i|y))p(y)$ ，但是此时 $p(x_i|y)$ 为多项式分布。

模型的参数如下：

$$\begin{aligned}\Phi_y &= p(y) \\ \Phi_{k|y=1} &= p(x_j = k|y = 1)\end{aligned}$$

式中， k 为单词表索引号， j 为单词在邮件中的位置。上式对任意的位置 j 均成立，即第 k 个单词出现的概率与位置 j 无关。

$$\Phi_{k|y=0} = p(x_j = k|y = 0)$$

若训练样本集为： $\{(x^{(i)}, y^{(i)}); i = 1, 2, \dots, m\}$ ，其中， m 为训练样本集数量，则每个样本可以表示为：

$$x^{(i)} = \{x_1^{(i)}, x_2^{(i)}, \dots, x_{n_i}^{(i)}\}$$

式中，上标 i 代表第 i 个样本，下标代表单词的位置，邮件的长度或称特征向量维度为 n_i 。

$$\begin{aligned}\mathcal{L}(\Phi, \Phi_{k|y=0}, \Phi_{k|y=1}) &= \prod_{i=1}^m p(x^{(i)}, y^{(i)}) \\ &= \prod_{i=1}^m \left(\prod_{j=1}^{n_i} p(x_j^{(i)}|y; \Phi_{k|y=0}, \Phi_{k|y=1}) \right) p(y^{(i)}; \Phi_y)\end{aligned}$$

在这个式子的推导中，用到了联合概率分布与条件概率分布的一个公式：

$$p(y|x) = \frac{p(x, y)}{p(x)}$$

这是标准的公式，可以将 x 和 y 的位置互换，得到以下公式：

$$p(x|y) = \frac{p(x, y)}{p(y)}$$

则有：

$$p(x, y) = p(x|y)p(y)$$

所以上式中右侧可以表示为:

$$\begin{aligned} p(x^{(i)}, y^{(i)}) &= p(x^{(i)} | y^{(i)}) p(y^{(i)}) \\ &= \left(\prod_{j=1}^{n_i} p(x_j^{(i)} | y^{(i)}) \right) p(y^{(i)}) \end{aligned}$$

因为第 i 封邮件由 n_i 个单词组成, 彼此间独立, 所以上式可以用连乘的方式求出其概率。将这个式子代入 $\mathcal{L}(\phi, \phi_{k|y=0}, \phi_{k|y=1}) = \prod_{i=1}^m p(x^{(i)}, y^{(i)})$, 就可以得到多项式事件模型似然函数的公式了。

我们可以按照最大似然函数方法求出模型参数, 如下:

$$\phi_{k|y=1} = \frac{\sum_{i=1}^m \sum_{j=1}^{n_i} 1\{x_j^{(i)} = k \text{ and } y^{(i)} = 1\}}{\sum_{i=1}^m 1\{y^{(i)} = 1\} n_i}$$

我们先来看分子部分, 在大括号里面, 对每个邮件的每个位置, 如果是单词 k 且该邮件是垃圾邮件, 则累加此样本, 并依次处理所有的样本。如果某个邮件是垃圾邮件, 分母部分则累加该邮件长度 n_i 。

$$\phi_{k|y=0} = \frac{\sum_{i=1}^m \sum_{j=1}^{n_i} 1\{x_j^{(i)} = k \text{ and } y^{(i)} = 0\}}{\sum_{i=1}^m 1\{y^{(i)} = 0\} n_i}$$

在分子部分的大括号里面, 对每个邮件的每个位置, 如果是单词 k 且该邮件不是垃圾邮件, 则累加此样本, 并依次处理所有的样本。如果某个邮件不是垃圾邮件, 分母部分则累加该邮件长度 n_i 。

$$\phi_y = \frac{\sum_{i=1}^m 1\{y^{(i)} = 1\}}{m}$$

如果考虑到拉普拉斯平滑, 则上述公式变为:

$$\begin{aligned} \phi_{k|y=1} &= \frac{\sum_{i=1}^m \sum_{j=1}^{n_i} 1\{x_j^{(i)} = k \text{ and } y^{(i)} = 1\} + 1}{\sum_{i=1}^m 1\{y^{(i)} = 1\} n_i + V} \\ \phi_{k|y=0} &= \frac{\sum_{i=1}^m \sum_{j=1}^{n_i} 1\{x_j^{(i)} = k \text{ and } y^{(i)} = 0\} + 1}{\sum_{i=1}^m 1\{y^{(i)} = 0\} n_i + V} \end{aligned}$$

式中, V 为单词表内总的单词数。

以上就是朴素贝叶斯模型的主要内容, 由于朴素贝叶斯模型比较简单, 而且在通常情况下其分类性能非常好, 因此我们通常先尝试这个方法, 看看利用它能不能取得满意的效果, 如果它不能解决问题我们再使用其他复杂的自然语言处理技术。

第 13 章

支撑向量机

在所有机器学习算法中，支撑向量机可以说是最受人关注的一种机器学习算法。它不仅具有线性模型的简单性，而且可以处理非线性分类问题。事实上，在深度学习复兴之前，研究人员关注的重点一直是支撑向量机，神经网络基本处于边缘化的位置。即使在深度学习大热的现在，支撑向量机也被公认为最有效的分类器之一。在 MNIST 手写数字识别中，在没有经过训练样本集变形处理的情况下，其在测试样本集上的误差可以达到 0.56%，接近于人工识别水平。而神经网络，以多层感知器模型为例，不经过训练样本集变形处理，其在测试样本集上的误差率为 1.6%；即使经过训练样本集变形处理，其误差率也仅能达到 0.7%。而且，神经网络的复杂度和训练周期，都比支撑向量机长得多。

本章首先介绍支撑向量机的理念，再介绍最优分隔器和优化问题，然后介绍核函数，接着介绍完整的支撑向量机算法，最后介绍支撑向量机的非线性分类问题的扩展。

13.1 支撑向量机概述

支撑向量机算法在数学形式上还是比较复杂的。但是从直观上来看，其指导思想还是很好理解的。支撑向量机主要面对两类别分类问题，类别标签为 -1、1，其核心思想就是在空间中找到一个超平面，使无论是 -1 类别还是 1 类别的样本点，到这个平面的距离都是最远的。这样说比较抽象，我们来举一个具体的例子，由于多维问题比较难以表示，先以一个二维的问题为例，大家可以把下面例子中的直线理解为多维问题中的超平面。

假设我们在做肿瘤诊断，其中只有两个特征：肿瘤大小 x_1 和年龄 x_2 ，阳性（肿瘤患者）用 \times 表示，阴性（非肿瘤患者）用 \bullet 表示。我们可以用各种直线来分隔阴性和阳性样本，但是如图 13.1 所示的直线无疑是最佳的分隔线，因为其离阴性和阳性都很远，用它可以很

清楚地判断各个样本是阴性还是阳性，可以非常有把握地下结论。

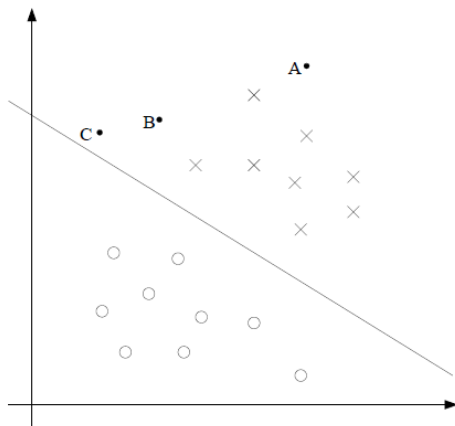


图 13.1 支撑向量机的直观理解

支撑向量机算法的目标就是发现这条直线，使我们能对所有样本进行正确分类。以上讨论的是二维问题，如果扩展到多维问题，上例中的直线就变成了超平面，但是找到超平面来辅助分类的原理是不变的。

13.1.1 函数间隔和几何间隔

在支撑向量机中，找到距离正、负样本都最远的超平面有助于对样本进行分类，所以需要先定义样本点到超平面的距离。

假设样本共有两类，标签 y 的定义： $y \in \{-1, 1\}$ 。

假设函数定义为： $h_{w,b}(x) \in \{-1, 1\}$ ，其形式为：

$$h_{w,b}(x) = g(z) = \begin{cases} 1, & z \geq 0 \\ -1, & z < 0 \end{cases}$$

在这里认为假设函数为线性函数，则有：

$$h_{w,b}(x) = g(\mathbf{w}^T \mathbf{x} + b) \quad \mathbf{w} \in \mathbb{R}^n, \mathbf{x} \in \mathbb{R}^n$$

有了上述规定之后，对于每个训练样本： $(\mathbf{x}^{(i)}, y^{(i)})$, $i \in \{1, 2, \dots, m\}$ 就可以定义函数距离 (Functional Margin)：

$$\hat{y}^{(i)} = y^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b)$$

根据支撑向量机的定义，我们希望 $y^{(i)} = 1$ 时， $\mathbf{w}^T \mathbf{x}^{(i)} + b \gg 0$ ，这时判断的依据就很充分；同理，当 $y^{(i)} = -1$ 时， $\mathbf{w}^T \mathbf{x}^{(i)} + b \ll 0$ ，这时判断依据也很充分。总之，就是希望函数距离 $\hat{y}^{(i)}$ 尽量大。

如果 $y^{(i)} = 1$ 时样本为正类别，在正确的情况下，应该有 $\mathbf{w}^T \mathbf{x}^{(i)} + b > 0$ ，但是如果参数

(w,b) 选择不合适， $w^T x^{(i)} + b < 0$ ，此时 $\hat{y}^{(i)} = y^{(i)}(w^T x^{(i)} + b) < 0$ 。由此可以看出，当函数距离小于 0 时，证明我们的判断是错误的。

如果 $y^{(i)} = -1$ 时样本为负类别，在正确的情况下，应该有 $w^T x^{(i)} + b < 0$ ，但是如果参数 (w,b) 选择不合适， $w^T x^{(i)} + b > 0$ ，此时 $\hat{y}^{(i)} = y^{(i)}(w^T x^{(i)} + b) < 0$ 。由此可以看出，当函数距离小于 0 时，证明我们的判断是错误的。

定义了每个样本的函数距离之后，可以定义针对整个样本集的函数距离：

$$\hat{y} = \min_{i=1,2,\dots,m} \hat{y}^{(i)}$$

这时，我们的任务就是使 \hat{y} 尽可能大。

下面我们来讨论几何距离，如图 13.2 所示。

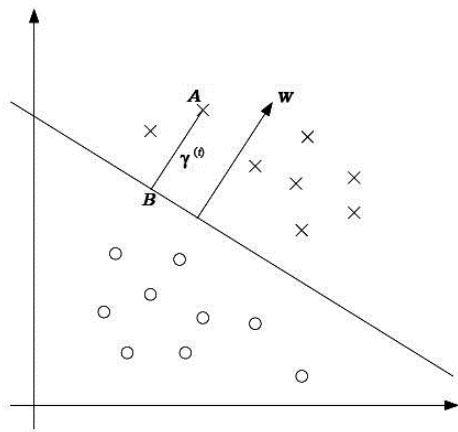


图 13.2 几何距离示意图

在图 13.2 中，假设 $x \in \mathbb{R}^2$ ，分别为 x_1 、 x_2 ，对应于图中的坐标轴，我们需要区分的是图中的点，可以通过超平面将样本集分为 1、-1 两类，超平面方程为：

$$w^T x + b = 0$$

展开后为：

$$w_1 x_1 + w_2 x_2 + b = 0$$

转化为直线方程为：

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{b}{w_2}$$

我们来看 A 点，代表训练样本 $x^{(i)}$ ，其类别为 $y^{(i)} = 1$ ，其到超平面的距离定义为 $\gamma^{(i)}$ ，为线段 AB 的长度。

根据相互垂直的直线斜率乘积为-1 的特性，可知其斜率应该为 $\frac{w_2}{w_1}$ ，可以求出超平面与 x_2 轴交点处的法线方程为：

$$\frac{x_2 - \left(-\frac{b}{w_2}\right)}{x_1 - 0} = \frac{x_2 + \frac{b}{w_2}}{x_1} = \frac{w_2}{w_1}$$

化简为标准直线方程为：

$$x_2 = \frac{w_2}{w_1}x_1 - \frac{b}{w_2}$$

其与两坐标轴交点为： $\left(0, -\frac{b}{w_2}\right)$ 和 $\left(\frac{w_1}{w_2^2}b, 0\right)$ ，则法向量为： $\left(\frac{w_1}{w_2^2}b, \frac{b}{w_2}\right)$ ，向量乘以常数其方向不变，我们将该向量乘以 $\frac{w_2^2}{b}$ ，就可以得到 (w_1, w_2) ，正好是向量 \mathbf{w} 。因此超平面法向量为 \mathbf{w} ，我们取单位法向量为： $\frac{\mathbf{w}}{\|\mathbf{w}\|}$ 。

下面求出 B 点坐标，定义 A 、 B 线段的距离为 $\gamma^{(i)}$ ，则有：

$$\mathbf{x}^{(i)} - \gamma^{(i)} \frac{\mathbf{w}}{\|\mathbf{w}\|}$$

因为 B 点在超平面上必须满足超平面方程，则有：

$$\mathbf{w}^T \left(\mathbf{x}^{(i)} - \gamma^{(i)} \frac{\mathbf{w}}{\|\mathbf{w}\|} \right) + b = 0$$

将上式展开得：

$$\mathbf{w}^T \mathbf{x}^{(i)} + b = \gamma^{(i)} \frac{\mathbf{w}^T \mathbf{w}}{\|\mathbf{w}\|} = \gamma^{(i)} \|\mathbf{w}\|$$

因为 $\mathbf{w}^T \mathbf{w} = \|\mathbf{w}\|^2$ ，所以可以解出 $\gamma^{(i)}$ 为：

$$\gamma^{(i)} = \left(\frac{\mathbf{w}}{\|\mathbf{w}\|} \right)^T \mathbf{x}^{(i)} + \frac{b}{\|\mathbf{w}\|}$$

以上推导是以 $y^{(i)} = 1$ 为基础的，当 $y^{(i)} = -1$ 时，就变为：

$$\gamma^{(i)} = y^{(i)} \left(\left(\frac{\mathbf{w}}{\|\mathbf{w}\|} \right)^T \mathbf{x}^{(i)} + \frac{b}{\|\mathbf{w}\|} \right)$$

上式就是针对单个训练样本的几何距离公式。我们可以看到，如果 $\|\mathbf{w}\| = 1$ ，则几何距离与函数距离相等。当 $w \geq 2w$, $b \geq 2b$ 时，几何距离不变，这个性质会在我们后面的公式推导中用到。

下面我们来定义在整个训练样本集上的几何距离：

$$\gamma = \min_{i=1,2,\dots,m} \gamma^{(i)}$$

13.1.2 最优距离分类器

给定训练样本集 S ，我们需要找到一个能够分隔正、负样本的超平面，使在该训练样本集上定义的几何距离达到最大值。这个问题实际上是分两步进行的，首先找到所有样本几何距离的最小值，然后求几何距离最大的情况。

实际上这个问题就转化为如下所示的优化问题：

$$\begin{aligned} \max_{\gamma, \mathbf{w}, b} \quad & \gamma \\ \text{s.t.} \quad & \gamma^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) \geq \gamma, \quad i = 1, 2, \dots, m \text{ 且 } \|\mathbf{w}\| = 1 \end{aligned} \quad (1)$$

式（1）表示调整参数 \mathbf{w}, b ，使得几何距离 γ 取得最大值。

但是在式（1）的优化问题中， $\|\mathbf{w}\| = 1$ 使权值位于单位球上，实际上是一个非凸的问题，而非凸问题具有多个局部最小值点，是很难找到全局最优解的。

为了去掉 $\|\mathbf{w}\| = 1$ 的限制条件，可以考虑利用优化函数距离来取代优化几何距离，上面的问题可以转化为：

$$\begin{aligned} \min_{\gamma, \mathbf{w}, b} \quad & \frac{\hat{\gamma}}{\|\mathbf{w}\|} \\ \text{s.t.} \quad & \gamma^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) \geq \hat{\gamma}, \quad i = 1, 2, \dots, m \end{aligned} \quad (2)$$

虽然去掉了 $\|\mathbf{w}\| = 1$ 的限制条件，这个问题同样是一个非凸优化问题，还是无法求出最优解。

因为放缩 \mathbf{w}, b 不改变几何距离大小，所以可以设 $\hat{\gamma} = 1$ ，则优化问题就变成优化 $\frac{\hat{\gamma}}{\|\mathbf{w}\|} = \frac{1}{\|\mathbf{w}\|}$ ，可以转化为求 $\|\mathbf{w}\|$ 的最小值，所以优化问题可以转化为：

$$\begin{aligned} \min_{\gamma, \mathbf{w}, b} \quad & \frac{1}{2} \|\mathbf{w}\|^2 \\ \text{s.t.} \quad & \gamma^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) \geq 1, \quad i = 1, 2, \dots, m \end{aligned} \quad (3)$$

这就是最优距离分类器模型。我们将优化问题变为一个二次方优化问题，同时只有线性的限制条件，可以通过 QP 算法求出最优解。

13.2 拉格朗日对偶

下面我们来讨论有限制条件的优化问题，如下：

$$\min_{\mathbf{w}} f(\mathbf{w})$$

$$\text{s.t. } h_i(\mathbf{w}) = 0, \quad i = 1, 2, \dots, \ell$$

上式表示以 \mathbf{w} 为参数，求 $f(\mathbf{w})$ 的最小值。我们定义拉格朗日算子：

$$\mathcal{L}(\mathbf{w}, \beta) = f(\mathbf{w}) + \sum_{i=1}^{\ell} \beta_i h_i(\mathbf{w})$$

式中， β_i 为拉格朗日乘数，为了求出参数 \mathbf{w} 和拉格朗日乘数 β ，可以分别设置拉格朗日算子对 \mathbf{w}, β 的偏导数为 0：

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}_i} = 0; \quad \frac{\partial \mathcal{L}}{\partial \beta_i} = 0$$

解出 \mathbf{w}, β 即可。

下面我们来定义主优化问题（Primal Optimization Problem）：

$$\begin{aligned} & \min_{\mathbf{w}} f(\mathbf{w}) \\ & \text{s.t. } g_i(\mathbf{w}) \leq 0, \quad i = 1, 2, \dots, k \\ & \quad h_i(\mathbf{w}) = 0, \quad i = 1, 2, \dots, \ell \end{aligned}$$

为了解决这个优化问题，需要定义通用拉格朗日算子：

$$\mathcal{L}(\mathbf{w}, \alpha, \beta) = f(\mathbf{w}) + \sum_{i=1}^k \alpha_i g_i(\mathbf{w}) + \sum_{i=1}^{\ell} \beta_i h_i(\mathbf{w})$$

式中， α_i, β_i 为拉格朗日乘数。我们定义为：

$$\theta_P = \max_{\alpha, \beta: \alpha_i \geq 0} \mathcal{L}(\mathbf{w}, \alpha, \beta)$$

可以很容易得到：

$$\theta_P(\mathbf{w}) = \begin{cases} f(\mathbf{w}), & \mathbf{w} \text{ 满足主优化限制条件} \\ \infty, & \mathbf{w} \text{ 不满足主优化限制条件} \end{cases}$$

主优化问题为：

$$p^* = \min_{\mathbf{w}} \theta_P = \min_{\mathbf{w}} \max_{\alpha, \beta: \alpha_i \geq 0} \mathcal{L}(\mathbf{w}, \alpha, \beta)$$

下面用一种不同的方法来定义这个优化问题，我们定义对偶优化：

$$\theta_D(\alpha, \beta) = \min_{\mathbf{w}} \mathcal{L}(\mathbf{w}, \alpha, \beta)$$

对偶优化问题可以定义为：

$$d^* = \max_{\alpha, \beta: \alpha_i \geq 0} \theta_D(\alpha, \beta) = \max_{\alpha, \beta: \alpha_i \geq 0} \min_{\mathbf{w}} \mathcal{L}(\mathbf{w}, \alpha, \beta)$$

我们看到，主优化和对偶优化问题非常相似，只是求最大值和最小值的顺序不同。我们可以证明， $\max \min \leq \min \max$ ，所以对偶优化和主优化关系为：

$$d^* = \max_{\alpha, \beta: \alpha_i \geq 0} \min_{\mathbf{w}} \mathcal{L}(\mathbf{w}, \alpha, \beta) \quad \min_{\mathbf{w}} \max_{\alpha, \beta: \alpha_i \geq 0} \mathcal{L}(\mathbf{w}, \alpha, \beta) = p^*$$

在某些条件下，可以取对偶优化值与主优化值相等，即： $d^* = p^*$ 。

如果 f 和 g_i 是凸函数， h_i 是线性仿射函数，则存在如下一组参数： $\mathbf{w}^*, \alpha^*, \beta^*$ ， \mathbf{w}^* 为主优化的解， α^*, β^* 为对偶优化的解，且 $d^* = p^* = \mathcal{L}(\mathbf{w}^*, \alpha^*, \beta^*)$ ，并且满足下列性质：

$$\frac{\partial}{\partial \mathbf{w}_i} \mathcal{L}(\mathbf{w}^*, \alpha^*, \beta^*) = 0, \quad i = 1, 2, \dots, n \quad ①$$

$$\frac{\partial}{\partial \beta_i} \mathcal{L}(\mathbf{w}^*, \alpha^*, \beta^*) = 0, \quad i = 1, 2, \dots, \ell \quad ②$$

$$\alpha_i^* g_i(\mathbf{w}^*) = 0, \quad i = 1, 2, \dots, k \quad ③$$

$$g_i(\mathbf{w}^*) \leq 0, \quad i = 1, 2, \dots, k \quad ④$$

$$\alpha_i^* > 0, \quad i = 1, 2, \dots, k \quad ⑤$$

这些性质即 KKT 条件，如果 $\mathbf{w}^*, \alpha^*, \beta^*$ 满足 KKT 条件，则其同时为主优化和对偶优化的解。

式中的③式称为 KKT 对偶互补条件，如果 $\alpha_i^* > 0$ ，则 $g_i(\mathbf{w}^*) = 0$ ，这个条件在后面引出支撑向量时会用到。

13.3 最优分类器算法

我们在 13.1.2 节中定义的优化问题为：

$$\min_{\gamma, \mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|^2 \quad (4)$$

$$\text{s.t. } y^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) \geq 1, \quad i = 1, 2, \dots, m$$

我们定义：

$$g_i(\mathbf{w}) = -y^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) + 1 \leq 0 \quad (5)$$

根据 KKT 条件中的③，如果 $\alpha_i > 0$ ，因为 $\alpha_i g_i(\mathbf{w}^*) = 0$ ，所以必须有 $g_i(\mathbf{w}) = 0$ ，也就是 $y^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) = 1 = \hat{\gamma}$ ，即函数距离取得最小值，此时几何距离同样是最小值，如图 13.3 所示。

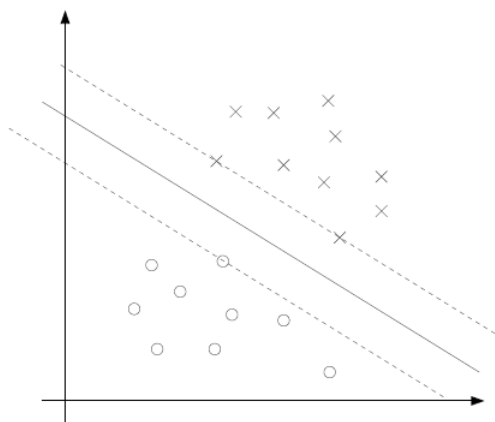


图 13.3 支撑向量示意

图中有两个 $y=1$ 的样本点和一个 $y=-1$ 的样本点，距离超平面几何距离最近，只有这些点上才有可能出现 $\alpha_i \gg 0$ 的情况，其他样本点必须满足 $\alpha_i = 0$ 的条件。也就是说，实际上样本集上绝大多数点 $\alpha_i = 0$ ，只有极少数点 $\alpha_i \gg 0$ ，这就大大降低了计算量。

定义向量的内积：

$$\langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle = (\mathbf{x}^{(i)})^T \mathbf{x}^{(j)} \quad (6)$$

定义主优化问题的拉格朗日算子为：

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^m \alpha_i [y^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + 1) - 1] \quad (7)$$

在优化问题中，只有 α_i 没有 β_i ，即只有不等于的限制条件，没有等于的限制条件。我们可以写出 KKT 条件，如下：

$$\frac{\partial}{\partial w_i} \mathcal{L}(\mathbf{w}^*, \alpha^*, \beta^*) = 0, \quad i = 1, 2, \dots, n \quad (1)$$

$$\alpha_i^* g_i(\mathbf{w}^*) = 0, \quad i = 1, 2, \dots, k \quad (3)$$

$$g_i(\mathbf{w}^*) \leq 0, \quad i = 1, 2, \dots, k \quad (4)$$

$$\alpha^* > 0, \quad i = 1, 2, \dots, k \quad (5)$$

为了求 $\frac{1}{2} \|\mathbf{w}\|^2$ 的极小值，将拉格朗日算子分别对 \mathbf{w} 和 b 求偏导，并令其为 0：

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}, b, \alpha) = \mathbf{w} - \sum_{i=1}^m \alpha_i y^{(i)} \mathbf{x}^{(i)} = 0 \quad (8)$$

可以解出：

$$\mathbf{w} = \sum_{i=1}^m \alpha_i y^{(i)} \mathbf{x}^{(i)} \quad (9)$$

对 b 求偏导，并令其值为 0：

$$\frac{\partial}{\partial b} \mathcal{L}(\mathbf{w}, b, \alpha) = - \sum_{i=1}^m \alpha_i y^{(i)} = 0 \quad (10)$$

在取得极小值的情况下，将上面两式代入式 (7)：

$$\begin{aligned} \mathcal{L}(\mathbf{w}, b, \alpha) &= \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^m \alpha_i [y^{(i)} (\mathbf{w}^T \mathbf{x}^{(i)} + 1) - 1] \\ &= \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i,j=1}^m y^{(i)} y^{(j)} \alpha_i \alpha_j (\mathbf{x}^{(i)})^T \mathbf{x}^{(j)} - b \sum_{i=1}^m \alpha_i y^{(i)} \\ &= \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i,j=1}^m y^{(i)} y^{(j)} \alpha_i \alpha_j \langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle \end{aligned} \quad (11)$$

上式为以 \mathbf{w} , b 为参数，求出使 $\mathcal{L}(\mathbf{w}, b, \alpha)$ 取极小值的情况，再以 α 为参数，求出上式的最大值，如下：

$$\begin{aligned} \max_{\alpha} \mathbf{w}(\alpha) &= \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i,j=1}^m y^{(i)} y^{(j)} \alpha_i \alpha_j \langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle \\ \text{s.t. } \alpha_i &\geq 0, \quad i = 1, 2, \dots, m \\ \sum_{i=1}^m \alpha_i y^{(i)} &= 0 \end{aligned} \quad (12)$$

由于我们是先求最小值再求最大值，由上节的定义，我们处理的是对偶优化问题，可以证明当前的优化问题满足 KKT 条件，同时满足对偶优化值与主优化问题值相等。

先根据式 (10) 求出 α_i ，然后根据式 (9) 求出 \mathbf{w} ，再根据下式求出 b 。这样就解决了这个最小距离分类器相关的优化问题。

在整个训练样本集上求出所有参数后，对一个新样本 \mathbf{x} 进行分类时，方法如下：

$$\mathbf{w}^T \mathbf{x} + b = \left(\sum_{i=1}^m \alpha_i y^{(i)} \mathbf{x}^{(i)} \right)^T \mathbf{x} + b = \sum_{i=1}^m \alpha_i y^{(i)} \langle \mathbf{x}^{(i)}, \mathbf{x} \rangle + b \quad (13)$$

在上式中，距离超平面最近的点对应的 $\alpha_i > 0$ ，其余的点对应 $\alpha_i = 0$ ，因此上式中绝大多数项均为 0，只有极少数点不为 0。另外，通过解对偶优化问题，在训练和实际运行时，均只需计算特征空间中特征向量的内积。

根据以上特点，我们将推导出支撑向量机算法，该算法的一大优点是可以高效地处理高维空间中的分类问题。有时为了分类方便，我们会将分类问题转换到高维空间来解决，这时支撑向量机就非常有竞争力了。顺便说一下，为什么将分类问题放到高维空间来解决呢？理论上的解释很抽象，我们在这里仅举一个例子，如果我们想分类只有 z 坐标不同的点，若在二维空间进行分类，这些点将是不可分的，但是到了三维空间之后，就可以很简单地根据 z 坐标完成分类问题了。

13.4 核方法

如果想要通过专家评分来预测公司的估值，可以使用线性回归方法，但是如果专家评分与公司估值之间不是线性关系，应用线性回归可能就不能取得很好的效果。这时可以通过引入多项式解决这一问题。假设引入 3 次方项，则特征向量将从一维变为三维，如下：

$$\mathbf{x} = [x] \Rightarrow \phi(\mathbf{x}) = \begin{bmatrix} x \\ x^2 \\ x^3 \end{bmatrix}$$

为了便于区分，将 \mathbf{x} 称为问题的属性，而 $\phi(\mathbf{x})$ 为特征， ϕ 为特征映射函数。

在支撑向量机应用中，基本上都是在特征向量空间中进行工作，而不是直接处理问题的属性，因此在上一节的公式中，我们需要将 \mathbf{x} 替换为 $\phi(\mathbf{x})$ 。

我们所研究的优化问题，是通过计算属性空间中两个向量的内积来实现的，可以将其变为特征空间中的向量内积。这时就需要将公式中的 $\langle \mathbf{x}, \mathbf{z} \rangle$ 变为 $\langle \phi(\mathbf{x}), \phi(\mathbf{z}) \rangle$ ，我们定义核函数：

$$K(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^T \phi(\mathbf{z})$$

因为在某些情况下，由于特征空间中向量的维度很高，无论是求特征向量，还是求特征向量的点积，都会比较困难，而通过核函数来计算往往会更加方便。

例如定义如下核函数：

$$\begin{aligned} K(\mathbf{x}, \mathbf{z}) &= \phi(\mathbf{x})^T \phi(\mathbf{z}) = (\mathbf{x}^T \mathbf{z})^2, \quad \mathbf{x}, \mathbf{z} \in \mathbb{R}^n \\ &= \left(\sum_{i=1}^n x_i z_i \right) \left(\sum_{j=1}^n x_j z_j \right) = \sum_{i=1}^n \sum_{j=1}^n x_i x_j z_i z_j \\ &= \sum_{i,j=1}^n (x_i x_j) (z_i z_j) \end{aligned}$$

如果 $n=3$ ，则：

$$K(\mathbf{x}, \mathbf{z}) = \sum_{i,j=1}^3 (x_i x_j)(z_i z_j) = \begin{bmatrix} x_1 x_1 \\ x_1 x_2 \\ x_1 x_3 \\ x_2 x_1 \\ x_2 x_2 \\ x_2 x_3 \\ x_3 x_1 \\ x_3 x_2 \\ x_3 x_3 \end{bmatrix}^T \begin{bmatrix} z_1 z_1 \\ z_1 z_2 \\ z_1 z_3 \\ z_2 z_1 \\ z_2 z_2 \\ z_2 z_3 \\ z_3 z_1 \\ z_3 z_2 \\ z_3 z_3 \end{bmatrix} = \phi(\mathbf{x})^T \phi(\mathbf{z})$$

从这个例子可以看出，计算 $\phi(\mathbf{x})$ 其复杂度为 $O(n^2)$ ，点积是两个 \mathbb{R}^{n^2} 的向量做点积，如果通过 $(\mathbf{x}^T \mathbf{z})^2$ 来计算，则复杂度为 $O(n)$ 。由此可以看出，直接计算核函数值将大大减少计算量，这种情况在高维的情况下更为显著。

下面再来看一个更一般的情况，假设核函数如下：

$$K(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^T \mathbf{z} + c)^2 = \sum_{i,j=1}^n (x_i x_j)(z_i z_j) + \sum_{i=1}^n (\sqrt{2c} x_i)(\sqrt{2c} z_i) + c^2$$

$$= \begin{bmatrix} x_1 x_1 \\ x_1 x_2 \\ x_1 x_3 \\ x_2 x_1 \\ x_2 x_2 \\ x_2 x_3 \\ x_3 x_1 \\ x_3 x_2 \\ x_3 x_3 \\ \sqrt{2c} x_1 \\ \sqrt{2c} x_2 \\ \sqrt{2c} x_3 \\ c^2 \end{bmatrix}^T \begin{bmatrix} z_1 z_1 \\ z_1 z_2 \\ z_1 z_3 \\ z_2 z_1 \\ z_2 z_2 \\ z_2 z_3 \\ z_3 z_1 \\ z_3 z_2 \\ z_3 z_3 \\ \sqrt{2c} z_1 \\ \sqrt{2c} z_2 \\ \sqrt{2c} z_3 \\ c^2 \end{bmatrix} = \phi(\mathbf{x})^T \phi(\mathbf{z})$$

我们可以将其推广为一般情况，设核函数为：

$$K(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^T \mathbf{z} + c)^d$$

确定了一个 $n+d$ 维 n 阶特征空间，对应 \mathbf{x} 的 d 阶多项式。但是由于核函数的存在，无须通过复杂度为 $O(n^d)$ 的计算求出 $\phi(\mathbf{x})$ ，只需通过复杂度为 $O(n)$ 的计算求出特征空间中向量的内积。

根据内积的定义，从直觉上来看，如果向量 $\phi(\mathbf{x})$ 和 $\phi(\mathbf{z})$ 越接近，其内积值越大；如果 $\phi(\mathbf{x})$ 和 $\phi(\mathbf{z})$ 越不同，如互为正交，则内积值将越小。综上所述，可以将核函数视为 $\phi(\mathbf{x})$ 和 $\phi(\mathbf{z})$ 相似程度的度量。

按照上面的理解，可以找到很多核函数。例如下面这个核函数：

$$K(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^T \phi(\mathbf{z}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{z}\|^2}{2\sigma^2}\right)$$

当 \mathbf{x} 和 \mathbf{z} 接近时, 核函数值接近 1; 当 \mathbf{x} 和 \mathbf{z} 相距较远时, 核函数值趋近 0。这个函数被称为高斯核函数, 可以处理无限维特征映射的问题。

那怎样判断一个函数是否可以作为核函数呢? 我们假设 K 是一个合法的核函数, 对一个有限的样本集:

$$\{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(m)}\}$$

我们定义核矩阵 $K \in \mathbb{R}^{m \times m}$, 其为 $m \times m$ 的方阵, 元素定义为:

$$K_{ij} = K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$$

如果 K 是合法核函数, 则必须有下式成立:

$$K_{ij} = K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}^{(j)}) = \phi(\mathbf{x}^{(j)})^T \phi(\mathbf{x}^{(i)}) = K(\mathbf{x}^{(j)}, \mathbf{x}^{(i)}) = K_{ji}$$

由此可以看出, 核矩阵必须为对称阵。另外, 令 $\phi_k(\mathbf{x})$ 为向量 $\phi(\mathbf{x})$ 第 k 维分量, 则对于任意向量 \mathbf{z} , 可以得出:

$$\begin{aligned} \mathbf{z}^T K \mathbf{z} &= \sum_i \sum_j z_i K_{ij} z_j = \sum_i \sum_j z_i \phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}^{(j)}) z_j = \sum_i \sum_j z_i \sum_k \phi_k(\mathbf{x}^{(i)}) \phi_k(\mathbf{x}^{(j)}) z_j \\ &= \sum_k \sum_i \sum_j z_i \phi_k(\mathbf{x}^{(i)}) \phi_k(\mathbf{x}^{(j)}) z_j \\ &= \sum_k \left(\sum_i z_i \phi_k(\mathbf{x}^{(i)}) \right)^2 \geq 0 \end{aligned}$$

由此我们得出结论: 核函数的充分必要条件为其对应的核矩阵为对称阵且元素值大于等于零。这实际上就是 Mercer 定理的内容。

下面我们来看几个核函数的应用实例。

13.5 非线性可分问题

到目前为止, 我们对支撑向量机的讨论均基于所研究的问题是线性可分的。但是如果所研究的问题是线性不可分的, 需要怎样处理呢? 虽然将问题通过特征映射到高维特征空间通常可以提高线性可分性, 但是并不能保证通过增加特征空间维度问题就一定线性可分。

还有一种现象, 如图 13.4 所示。

在左图中, 我们可以得到一个比较好的超平面, 可以很好地分隔开正、负样本集。但是当样本集的左上角多出一个额外的离群样本时, 如右图所示, 超平面就如实线所示, 从图中可以看出, 仅仅一个样本, 就使新的超平面与正、负样本集间的最小距离变得非常小, 增加了分类的不确定性, 显然这种情况是非常不好的, 我们应该想办法避免。

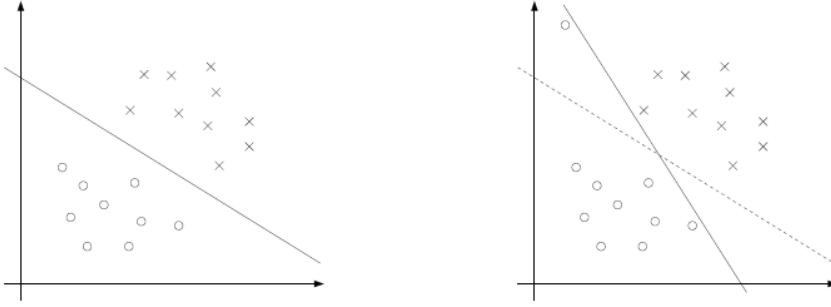


图 13.4 支撑向量机离群样本恶化的超平面示意图

针对以上两种现象，我们可以在优化问题上增加调整项 ℓ_1 ，如下：

$$\begin{aligned} \min_{\gamma, \mathbf{w}, b} \quad & \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^m \xi_i \\ \text{s.t.} \quad & y^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) \geq 1 - \xi_i, \quad i = 1, 2, \dots, m \\ & \xi_i \geq 0, \quad i = 1, 2, \dots, m \end{aligned} \quad (14)$$

在这里引入 ξ_i ，允许样本点函数距离小于 1， ξ_i 可以控制小于的程度，而这会使代价函数增加 $C\xi_i$ 。参数 C 用来协调取 $\|\mathbf{w}\|^2$ 最小值和样本点小于 1 的程度。

定义拉格朗日算子如下：

$$\begin{aligned} \mathcal{L}(\mathbf{w}, b, \xi, \alpha, r) = & \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^m \xi_i - \\ & \sum_{i=1}^m \alpha_i [y^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) - 1 + \xi_i] - \sum_{i=1}^m r_i \xi_i \end{aligned} \quad (15)$$

式中， $\alpha_i \geq 0$ 且 $r_i \geq 0$ 为拉格朗乘数。为了求出最小值，分别对 \mathbf{w} 、 b 求偏导，并令偏导为 0。

先对 \mathbf{w} 求偏导，并令其为 0：

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}, b, \xi, \alpha, r) = \mathbf{w} - \sum_{i=1}^m \alpha_i y^{(i)} \mathbf{x}^{(i)} = 0 \quad (16)$$

可以得到 \mathbf{w} 为：

$$\mathbf{w} = \sum_{i=1}^m \alpha_i y^{(i)} \mathbf{x}^{(i)} \quad (17)$$

再对 b 求偏导，并令其为 0：

$$\frac{\partial}{\partial b} \mathcal{L}(\mathbf{w}, b, \xi, \alpha, r) = - \sum_{i=1}^m \alpha_i y^{(i)} = 0 \quad (18)$$

可以得到:

$$\sum_{i=1}^m \alpha_i y^{(i)} = 0 \quad (19)$$

将式 (17) 和式 (19) 代入式 (15), 化简后可得:

$$W(\alpha) = \min_{\mathbf{w}, b} \mathcal{L}(\mathbf{w}, b, \xi, \alpha, r) = \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i,j=1}^m y^{(i)} y^{(j)} \alpha_i \alpha_j \langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle$$

优化问题可以变为:

$$\begin{aligned} \max_{\alpha} W(\alpha) &= \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i,j=1}^m y^{(i)} y^{(j)} \alpha_i \alpha_j \langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle \\ \text{s. t. } &0 \leq \alpha_i \leq C, \quad i = 1, \dots, m \\ &\sum_{i=1}^m \alpha_i y^{(i)} = 0 \end{aligned}$$

根据前面的讨论, 定义:

$$g_i(\mathbf{w}) = -y^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) + 1 \quad 0$$

此时 KKT 条件为:

$$\frac{\partial}{\partial w_i} \mathcal{L}(\mathbf{w}^*, \alpha^*, \beta^*) = 0, \quad i = 1, \dots, n \quad (1)$$

$$\alpha_i^* g_i(\mathbf{w}^*) = 0, \quad i = 1, \dots, k \quad (2)$$

$$g_i(\mathbf{w}^*) \leq 0, \quad i = 1, \dots, k \quad (3)$$

$$0 \leq \alpha_i^* \leq C, \quad i = 1, \dots, k \quad (4)$$

$$\alpha_i^* = 0 \Rightarrow y^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) = 1 \quad (5)$$

$$\alpha_i^* = C \Rightarrow y^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) = -1 \quad (6)$$

$$0 < \alpha_i^* < C \Rightarrow y^{(i)}(\mathbf{w}^T \mathbf{x}^{(i)} + b) = 0 \quad (7)$$

13.6 SMO 算法

在线性不可分情况下, 支撑向量机就难以处理上述问题了。在这一节中, 我们将讨论 SMO 算法来求解这一问题。

13.6.1 坐标上升算法

在机器学习中常用的学习算法有梯度下降算法和牛顿法，在这一节中，我们介绍一种新的算法——坐标上升算法。

假设要解决如下无限制条件的优化问题：

$$\max_{\alpha} W(\alpha_1, \alpha_2, \dots, \alpha_m)$$

为了解决这一优化问题，可以采用如下梯度上升算法，循环直到收敛为止：

对于 $i=1, \dots, m$ ：

$$\alpha_i = \operatorname{argmax}_{\hat{\alpha}_i} W(\alpha_1, \dots, \alpha_{i-1}, \hat{\alpha}_i, \alpha_{i+1}, \dots, \alpha_m)$$

在上述循环中的最内层，每次固定所有参数的值，除 α_i 外。我们将 α_i 视为变量，求 W 为最大值时 α_i 对应的取值，顺序为 $\alpha_1, \alpha_2, \dots, \alpha_m, \alpha_1, \alpha_2, \dots, \alpha_m$ ，所以这种算法称为坐标上升算法。当所有参数 $\alpha_1, \alpha_2, \dots, \alpha_m$ 不再改变时，就认为达到收敛条件，停止最外层循环，这时就找到了 W 取最大值时所有参数的值，也可以确定 W 的最大解了。

当然，在实际应用中，可以不必完全按照 $\alpha_1, \alpha_2, \dots, \alpha_m$ 的顺序，也可以根据使 W 取最大值有最大可能的参数进行调整，这样算法的收敛速度会更快一些。但是标准的坐标上升算法的收敛速度通常还是非常快的。

13.6.2 SMO 算法详解

支撑向量机对应的优化问题（对偶优化）为：

$$\max_{\alpha} W(\alpha) = \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i,j=1}^m y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle \quad (1)$$

$$\text{s.t. } 0 \leq \alpha_i \leq C, \quad i = 1, 2, \dots, m \quad (2)$$

$$\sum_{i=1}^m \alpha_i y^{(i)} = 0 \quad (3) \quad (20)$$

这里不能直接应用坐标上升算法。由于有式（20）的限制条件，如果固定 $\alpha_2, \dots, \alpha_m$ 的值不变，只取 α_1 来优化 $W(\alpha)$ 的话，得出的 α_1 值很可能会违反式（20）中③的限制条件。实际上，如果固定 $\alpha_2, \dots, \alpha_m$ 的值， α_1 的值也就确定了，如下：

$$\alpha_1 y^{(1)} = - \sum_{i=2}^m \alpha_i y^{(i)}$$

将上式两边同时乘以 $y^{(1)}$ ，且 $y^{(1)} \in \{-1, 1\}$ ，可得：

$$\alpha_1 = -y^{(1)} \sum_{i=2}^m \alpha_i y^{(i)}$$

为了解决这一问题，采用坐标上升算法时必须同时改变两个参数的值。下面以同时改变 α_1 和 α_2 为例，推导坐标上升算法。

现在固定 $\alpha_3, \dots, \alpha_m$ 的值，通过改变 α_1, α_2 的值求出 $W(\alpha)$ 的最大值。根据式(20)中③可知：

$$\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = - \sum_{i=3}^m \alpha_i y^{(i)}$$

为了简洁起见，定义：

$$\zeta = - \sum_{i=3}^m \alpha_i y^{(i)}$$

则：

$$\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = \zeta$$

如果以 α_1, α_2 为坐标轴，根据式(20)中②可知， α_1, α_2 的取值范围在 $[0, C] \times [0, C]$ 的矩形区域内，为了满足所有限制条件必然需要 $L \leq \alpha_2 \leq H$ ，如图 13.5 所示。

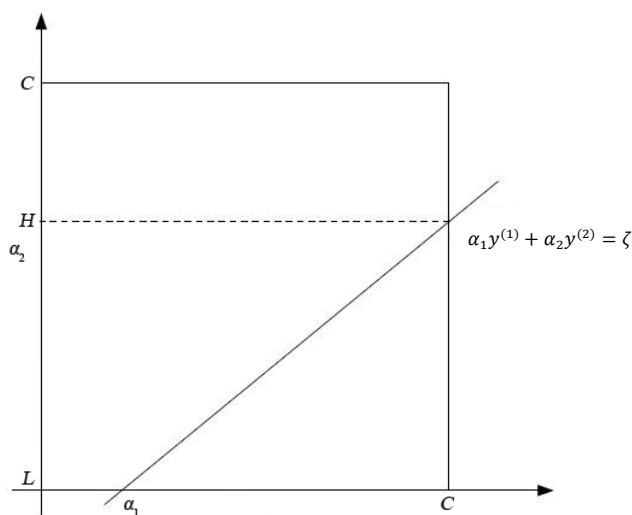


图 13.5 α_1, α_2 取值范围示意图

可以将 α_1 用 α_2 来表示，则：

$$\alpha_1 = (\zeta - \alpha_2 y^{(2)}) y^{(1)}$$

则 W 函数就可以表示为：

$$W(\alpha_1, \alpha_2, \dots, \alpha_m) = W\left((\zeta - \alpha_2 y^{(2)})y^{(1)}, \alpha_2, \dots, \alpha_m\right)$$

根据上式可知， W 可以表示为 α_2 的平方函数，可以视为以 a, b, c 为参数的形式：

$$\alpha_2 = a\alpha_2^2 + b\alpha_2 + c$$

为了求出 W 的最大值，对 W 进行求导，并令导数为 0：

$$\frac{\partial}{\partial \alpha_2} W\left((\zeta - \alpha_2 y^{(2)})y^{(1)}, \alpha_2, \dots, \alpha_m\right) = 0$$

可以解出 α_2^{raw} ，但是这个值可能在 $[L, H]$ 的范围之外，因此可以按照下面的公式得出最终 α_2 的值：

$$\alpha_2^{\text{new}} = f(x) = \begin{cases} H, & \alpha_2^{\text{raw}} > H \\ \alpha_2^{\text{raw}}, & L \leq \alpha_2^{\text{raw}} \leq H \\ L, & \alpha_2^{\text{raw}} < L \end{cases} \quad (21)$$

第五部分 深度学习平台 API

- Python Web 编程
- 深度学习云平台

第 14 章

Python Web 编程

在前面的章节中，我们向大家介绍了深度学习的主要算法，并且以 Theano 框架或 Python+Numpy 为例，给出了实际运行的例子。如果是用于研究和学习，这些知识就已经足够了。但是如果是想在实际中应用，那还缺很关键的一环，就是向外界提供深度学习服务的接口。无论是面向公司内部项目，还是面向公众项目，采用接口形式提供深度学习服务都要更合理一些。

在本章中，我们将带领大家利用 Python 做一个深度学习服务的 Web 接口，最终用户将通过接口来调用深度学习服务。我们将从头开发一个小型 Web 接口服务器，其支持 AJAX 请求、数据持久化、用户认证、任务队列、数据集文件上传等功能。我们将把这个 Web 服务器与我们的深度学习服务相结合，开发一个图像识别系统，用户可以输入一个手写数字图片，通过服务得到识别的结果。通过这个简单的实例，使读者能够理解深度学习平台的构建方式。

14.1 Python Web 开发环境搭建

Python 进行 Web 开发有很多方式，其中最常用的莫过于采用强大的 Django 框架。然而这里并不建议采用这一框架，原因主要有以下几个方面：第一，Django 框架太过庞大，里面提供了面面俱到的功能，作为一个接口服务器，我们只会使用到 Django 中不到 1% 的功能，因此引入一个庞大的服务器开发框架可能是得不偿失的；第二，我们的接口服务器虽然功能相对简单，但是需要面对不同问题，比如需要处理的任务有可能很长，任务数据集也可能很大，因此任务可能需要队列，需要异步方式通知调用者服务的运行结果，因为这些内容不是典型的 Web 应用特征，Django 框架对这些方面提供的支持不是特别友好，扩

展 Django 在这方面的功能可能比重新开发还要困难；第三，Django 框架是一个老牌框架，在 Python 2 时期就已经非常流行，而且长时间只支持 Python 2 系统开发，只是最近才正式支持 Python 3 系统，因此 Django 中很多方面都是以 Python 2 为基础的，会有一些设计不完善的地方。综上所述，我们觉得做这个深度学习服务接口服务器，不妨重新发明一下轮子，用 Python 3 实现一个轻量级的深度学习服务接口服务器。当然，如果完全从头开发一个 Web 服务器，工作量将非常大，而且也很难保证质量，毕竟开发一个高质量的大容量高并发 Web 服务器，即使使用 Python 3 这种高级语言，也不是一件特别容易的事情。因此我们还需要借助一些轻量级框架。在这里，我们所选择的框架是 CherryPy。

14.1.1 CherryPy 框架

CherryPy 号称最简单的 Web 框架，与 Django 等流行框架不同，CherryPy 并没有提供 Web 开发中包罗万象的技术，只提供了一个满足基本 HTTP 的 Web 服务器框架，对于数据操作、缓存操作、WebSocket 支持、HTML 模板技术等，都需要开发者自己选择合适的技术来做，框架本身并不提供默认的选择。这样看来，与 Django 等包罗万象的强大主流框架相比，似乎 CherryPy 在做 Web 开发方面要有先天的劣势。但是事实却不完全如此，Django 等框架非常强大不假，但是对于我们的接口服务器来说，它可能太重了，例如 HTML 模板技术、ORM 技术等，我们根本就不需要，而我们需要的 WebSocket 技术它又可能没提供，而且异步任务和消息队列都不是 Django 等的强项。而我们在 CherryPy 框架中集成这些技术却是相当简单的事情，只需要利用 pip 命令安装一个包，并用 import 命令引入这个模块就可以正常使用了，所以 CherryPy 在这种接口服务器开发中还是一个不错的选择。

CherryPy 是一个有着 10 年以上历史的 Web 框架，目前既有超小型系统在使用，也有大型关键应用在使用。CherryPy 一个很大的好处就在于，其支持开发者像开发普通应用一样开发 Web 应用，可以自由使用面向对象编程或函数式编程风格。

CherryPy 具有以下特性：

- ☐ 符合 HTTP/1.1 协议，支持基于 WSGI 的线程池模型。
- ☐ 可以同时监听多个端口，可以独立运行。
- ☐ 强大的配置管理系统，使开发和部署都非常方便。
- ☐ 支持利用插件系统进行扩展。
- ☐ 内置支持缓存、Session、认证、静态内容（CSS、JS 等）。
- ☐ 支持性能调优和覆盖性测试。

14.1.2 CherryPy 安装

有多种方法可以安装 CherryPy，最简单的方式为采用 pip，先启动虚拟环境，然后进行安装：

```
pip install cherrypy
```

这条命令会将 CherryPy 安装到虚拟环境的 lib 目录下，安装完成之后就可以通过 `import cherrypy` 直接使用了。

还可以下载 CherryPy 源码，在 Python 的虚拟环境下，进入 CherryPy 源码目录，运行如下命令：

```
python setup.py
```

这条命令同样会将 CherryPy 安装到虚拟环境的 lib 目录下，安装完成之后就可以通过 `import cherrypy` 直接使用了。

也可以不安装，直接使用 CherryPy 源码，我们将采用这种方式。先到 CherryPy 的 GitHub 主页找到项目的 Git 源，然后在 Python 虚拟环境下进入一个空目录，运行如下命令：

```
wget https://github.com/cherrypy/cherrypy.git
```

执行完成后，会在当前目录下建立一个 `cherrypy` 目录，进入该目录就可以看到 CherryPy 的源码了。

14.1.3 测试 CherryPy 安装是否成功

下面先来写一个最简单的小程序，来测试一下 CherryPy 是否安装成功，代码如下（注意：请确定在 `cherrypy` 源码根目录下）：

```
1 import sys
2 sys.path.append('./cherrypy')
3 import cherrypy
4
5 class HelloWorld(object):
6     @cherrypy.expose
7     def index(self):
8         return 'CherryPy安装成功'
9
10 if __name__ == '__main__':
11     cherrypy.quickstart(HelloWorld(), '/')
```

运行结果如下：

```
[01/Jan/2017:08:35:26] ENGINE Listening for SIGHUP.
[01/Jan/2017:08:35:26] ENGINE Listening for SIGUSR1.
[01/Jan/2017:08:35:26] ENGINE Listening for SIGTERM.
[01/Jan/2017:08:35:26] ENGINE Bus STARTING
CherryPy Checker:
The Application mounted at '' has an empty config.

[01/Jan/2017:08:35:26] ENGINE Started monitor thread '_TimeoutMonitor'.
[01/Jan/2017:08:35:26] ENGINE Started monitor thread 'Autoreloader'.
[01/Jan/2017:08:35:27] ENGINE Serving on http://127.0.0.1:8080
[01/Jan/2017:08:35:27] ENGINE Bus STARTED
127.0.0.1 - - [01/Jan/2017:08:35:41] "GET / HTTP/1.1" 200 20 "" "Mozilla/5.0 (X1
1; Ubuntu; Linux x86_64; rv:47.0) Gecko/20100101 Firefox/47.0"
```

在运行 CherryPy 程序的计算机上打开浏览器，在地址栏中输入 `http://localhost:8080`，可以看到如图 14.1 所示页面。

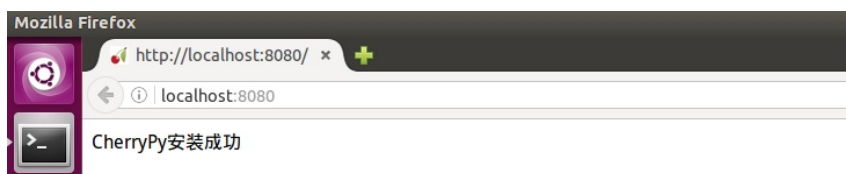


图 14.1 CherryPy 测试成功显示页面

读者如果看到上面的页面，就说明 CherryPy 安装成功了。但是这里还有几个遗留问题，如果我们在其他机器上打开浏览器输入 `http://serverIP:8080/`，则会提示找不到网页。另外，如果我们的 8080 端口被占用了，那么程序启动时就会出现问题。为了解决这些问题，我们将在下一节中向大家展示一个真正的最简 Web 服务器。

14.2 最简 Web 服务器

下面我们来看怎样用 CherryPy 框架做一个最简 Web 服务器。这个服务器需要向客户端展示网页内容，就像普通的 Web 服务器一样，可以显示 HTML 文件，并能处理页面中的 CSS 文件和 JS 文件，即要实现一个能够显示静态网站内容的服务器。

14.2.1 程序启动

在上一节中，我们的 Web 服务器仅能监听 localhost 的 8080 端口，现在希望其监听物理 IP 地址，同时我们可以指定端口。假设机器的物理 IP 地址为 192.168.1.16，希望监听 8090 端口。

在 CherryPy 中，这些配置信息被称为全局配置信息，可以通过 `cherrypy.config.update` 函数进行更新。在本例中，我们可以通过以下代码实现修改全局配置信息的目的：

```
cherrypy.config.update({'server.socket_host': '...', 'server.socket_port': ...})
```

实际上，除了可以设置全局配置信息，还可以设置应用特有的配置信息。通过这些配置信息，我们可以定制 Web 服务器的行为，代码如下：

```
1 import sys
2 sys.path.append('./cherrypy')
3 import cherrypy
4
5 class HelloWorld(object):
6     @cherrypy.expose
7     def index(self):
8         return 'Server start up'
9
10 if __name__ == '__main__':
11     cherrypy.config.update({
12         'server.socket_host': '192.168.1.16',
13         'server.socket_port': 8090,
14     })
15     cherrypy.quickstart(HelloWorld(), '/', {'/': {
16         }}
```

第 2 行：将 `cherrypy` 路径加入系统路径中。

第 3 行：引入 `cherrypy` 包，实际上，我们通过此行将 `./cherrypy` 目录加入系统目录，这时我们进行 `import` 操作，将在 `./cherrypy` 目录下查找 `cherrypy` 目录，这样就可以成功了。

第 5 行：定义 `HelloWorld` 类，就像平常开发普通应用时一样。

第 6 行：采用注解方式，指定下面的 `index` 方法是 `CherryPy` 的一个方法。通过注解方式指定代码的用途可以简化代码编写，这是近代编程的一大进步。

第 7 行：定义 `index` 方法，当用户请求没有任何路径时，会调用这一默认方法。

第 8 行：返回的字符串内容，将显示到用户的浏览器中。

第 10~14 行：改变 `CherryPy` 的全局配置，变为在物理 IP 地址上监听，端口为 8090。

第 15~16 行：调用 `cherrypy.quickstart` 启动 Web 服务器，参数如下。

- ❑ `class`：对应的实现类，请求将转发给该类处理。
- ❑ `path_patter`：路径模式，本例中就是所有的 URL。
- ❑ `config`：本应用的配置信息，在本例中不需要配置信息，所以是一个空的 `Object`。

14.2.2 显示 HTML 文件

上一节中的 Web 服务器已经可以在物理 IP 地址和任意端口上运行了，但是只是向用户浏览器输出 `Hello World`，也没有太大的实用价值，一般的 Web 服务器可以向用户显示静态 HTML 文件，所以也让我们的服务器拥有这一能力。首先，需要向用户提供 HTML 文件，然后为了正常显示 HTML 页面，还需要将页面中的 CSS、JS、图片文件正确地返回给浏览器，这时浏览器才能正确地显示一个 HTML 网页。我们将把这一过程分为两节来描述，本节先来讲怎么显示 HTML 文件，下一节将描述怎样向浏览器提供页面中需要的 CSS、JS、图片文件。

向用户端浏览器提供 HTML 内容有三种方法：第一种方法是将 HTML 内容写在程序中，通过字符串返回；第二种方法是如果采用 `AngularJS` 或 `ReactJS` 等前端技术，可以先将页面生成其对应的模板内容，然后将这些 JS 代码返回给浏览器，由浏览器解析运行，最后生成界面内容；第三种方法是将本地 HTML 文件内容提供给浏览器。

这三种方法各有利弊，在实际应用中需要根据自己的需要加以选择。

由于我们的接口服务器界面的工作量极小，因此选用第二种方法，即偏前端技术的方法，就不是特别适合了。第一种方法最大的优点就是快速，不用进行额外的文件读/写操作，效率会提高。若选择第三种方法，文件以 HTML 形式存在，便于编辑和调试。下面我们分别采用这两种方式显示纯 HTML 页面，然后再来讨论应该采用哪种方法。

我们来看怎样将 HTML 文件写在程序中直接提供给浏览器，代码如下：

```

1 import sys
2 sys.path.append('./cherry.py')
3 import cherrypy
4
5 class HelloWorld(object):
6     def __init__(self):
7         self.index_html = '''
8 <html>
9 <body>
10 Hello HTML page!
11 </body>
12 </html>
13 '''
14
15     @cherrypy.expose
16     def index(self):
17         return self.index_html
18
19 if __name__ == '__main__':
20     cherrypy.config.update({
21         'server.socket_host': '192.168.1.16',
22         'server.socket_port': 8090,
23     })
24     cherrypy.quickstart(HelloWorld(), '/', {'/': {
25         }})

```

这段程序与之前的程序并没有太大的区别，只对重点的区分部分进行讲解。

第 6 行：定义了 HelloWorld 类的构造函数。

第 7~13 行：定义属性 index_html 为字符型属性，值为 index.html 的内容，采用 “'''” 定义字符串内容，这样就不用在字符串内对 “'” 或 “'” 加转义符了。

第 17 行：返回内容时，直接返回 index_html 属性值，即上述 HTML 内容。

由上面的程序可以看出，将 HTML 内容直接写在源码文件中，以字符串形式出现，程序不需要做太大的修改，效率也会有所提高。但是这种方法的缺点还是不少的，首先，“'''” 字符串形式虽然免去了写转义字符，但是由于在字符串内，HTML 语法高亮显示肯定是没有了，直接用手写出正确的 HTML 难度有些大；其次，无论怎么排版，字符串定义与 HTML 代码之间总是会有风格不一致的地方，使代码显得不够整洁；最后，这里只有一个页面，可以直接将代码写在源码文件中，假设有成千上万个 HTML 内容时再这样操作，那加载时就不具备性能优势了。因此，在实际应用中，我们较少采用这种方式。

第三种方式是直接将 HTML 文件内容提供给浏览器。在下面的程序中，我们将用两种方式向浏览器提供 HTML 文件。第一种方式是直接通过文本文件读取，读出对应的 HTML 文件内容，以字符串形式返回给浏览器。第二种方式则是通过静态文件处理方式，将文件内容发送给浏览器。我们重点讲述第一种方式，第二种方式将在下一节重点讲述。

先来讲述读取 HTML 文件内容，并返回给浏览器的方法，代码如下：

```

1 import sys
2 sys.path.append('./cherry.py')
3 import cherrypy
4
5 class HelloWorld(object):
6     def __init__(self):
7         self.web_dir = '/home/osboxes/dev/wky/work/book/chp14/public/'
8
9     @cherrypy.expose
10     def index(self):
11         return 'Server start up'
12
13     @cherrypy.expose
14     def test(self):
15         fo = open(self.web_dir + 'test.html', 'r')

```



```

16         try:
17             html = fo.read()
18         finally:
19             fo.close()
20         return html
21
22 if __name__ == '__main__':
23     cherrypy.config.update({
24         'server.socket_host': '192.168.1.16',
25         'server.socket_port': 8090,
26     })
27     cherrypy.quickstart>HelloWorld(), '/', {'/': {}}})

```

第 6 行：先定义 HelloWorld 类的构造函数。

第 7 行：定义属性 web_dir，其定义 HTML 存放的根目录。

第 13、14 行：定义 cherrypy 新命令 test，访问的 URL 为 http://192.168.1.16:8090/test。

第 15 行：以只读方式打开 HTML 目录下的 test.html 文件。

第 17 行：读出 test.html 文件中的内容并将其赋给 html。

第 18 行：在 finally 里关闭文件。注意：打开文件必须要关闭，我们采用异常捕获方式，保证无论操作成功还是失败，都会关闭文件。

第 20 行：返回 HTML 文件的内容。

在/home/osboxes/dev/wky/work/book/chp14/目录下建立 public 目录，在其中添加 test.html 文件，代码如下：

```

1 <html>
2 <body>
3 中文：<br />
4 Hello test.html
5 </body>
6 </html>

```

14.2.3 静态内容处理

上一节中我们显示了一个最简单的 HTML 页面，但是在实际应用中，HTML 页面内有图片，有样式，还有 JS 文件，只有所有元素都正确，浏览器才能正确显示 HTML 页面。而目前我们的程序只能显示 HTML 文件信息，如果在其中加入图片、样式、JS 文件链接，浏览器将无法正确显示。下面我们就来解决这一问题，在页面中引入图片、加入样式，并通过引入 jQuery 加入交互内容。

先通过改变应用的配置信息让 CherryPy 处理静态内容，编辑 wky.conf 文件，内容如下：

```

1 [/]
2 tools.staticdir.root = '/home/osboxes/dev/wky/work/book/chp14'
3
4 [/public]
5 tools.staticdir.on = True
6 tools.staticdir.dir = 'public'

```

第 2 行：定义了配置静态文件的根目录。

第 4 行：定义 http://192.168.1.16:8090/public 下面的内容为静态内容。

第 5 行：定义对这个 URL 打开静态目录设置。

第 6 行：定义这个 URL 对应的静态文件存放目录，目录需要加上第 2 行定义的目录前缀。

接下来准备一些静态内容放到对应的目录中，进入/home/osboxes/dev/wky/work/book/chp14/public 目录，创建下列目录：

```
(wky) osboxes@osboxes:~/dev/wky/work/book/t$ mkdir js
(wky) osboxes@osboxes:~/dev/wky/work/book/t$ mkdir css
(wky) osboxes@osboxes:~/dev/wky/work/book/t$ mkdir images
```

然后下载 jQuery 和 jQuery UI，并将文件放入相应的目录下：

```
(wky) osboxes@osboxes:~/dev/wky/work/book/chp14/public/js$ wget https://code.jqu
ery.com/jquery-3.1.1.js
```

进入 css 目录，编辑一个简单的 CSS 文件 wky.css，如下：

```
1 /* red background */
2 .red_bg {
3     background-color: red;
4 }
```

上面的文件定义了一个背景颜色为红色的样式。

从网络上下载一张图片，并放到 images 目录下，将其名字改为 t1.jpg。

下面来看网络，还是 add.html，其有三个编辑框和一个等号按钮，当我们在前两个编辑框中输入值后，按“等号”按钮，就会将两个数相加的结果写入第三个编辑框中。因为这里有比较复杂的逻辑，所以将 add.html 作为一个 HTML5 文件的模板，引入了一些基本元素，代码如下：

```
1 <!DOCTYPE HTML>
2 <html lang="en">
3 <head>
4 <meta charset="utf-8">
5 <title>测试</title>
6 <meta name="viewport" content="width=device-width, initial-scale=1.0">
7 <meta name="description" content="">
8 <meta name="author" content="">
9 <script type="text/javascript" src="public/js/jquery-3.1.1.js"></script>
10 <link rel="stylesheet" type="text/css" href="public/css/wky.css">
11 <!--bootstrap-->
12 <script type="text/javascript" src="public/js/bootstrap.min.js"></script>
13 <link rel="stylesheet" type="text/css" href="public/js/bootstrap.min.css">
14 </head>
15 <body>
16 <div class="red_bg">
17     <span style="color: white;">红色背景</span>
18 </div>
19 
20 <br />
21 <input type="number" id="num1" placeholder="加数1" />
22 <span>+</span>
23 <input type="number" id="num2" placeholder="加数2" />
24 <input type="button" id="eqBtn" value="=" />
25 <input type="text" id="rst" placeholder="结果" readonly />
26
27 <script type="text/javascript">
28 $(document).ready(function() {
29     $("#eqBtn").bind("click", addNum);
30 });
31
32 function addNum() {
33     var num1 = $("#num1").val();
34     var num2 = $("#num2").val();
35     var rst = parseInt(num1) + parseInt(num2);
36     $("#rst").val("" + rst);
37 }
38 </script>
39
40 </body>
41 </html>
```

第 1 行：采用 HTML5 技术标准的文件头。

第 6 行：为了在移动设备上能正确显示页面，加入页面缩放机制，否则页面在手机等移动设备上将看起来特别小。

第 9 行：引入了 jquery-3.1.1.js 文件，实际上也可以引入 jquery-3.1.1.min.js 文件，这是文件的压缩形式，可以大幅减小文件的体积，但是其不便于调试，所以在开发阶段采用非压缩形式比较好。另外，其实引入 jQuery 可以选择直接引入在第三方 CDN 网络上的 jQuery 链接，这样不仅节省本机服务器资源，而且下载速度会更快，用户体验也会更好。这些可以在最终部署时进行考虑，在开发阶段可以采用本地 jquery-3.1.1.js 文件，这样可以方便调试工作。

第 10 行：引入 CSS 文件的标准语法。

第 16~18 行：显示红底白字的文本。

第 15 行：显示一张图片。

第 21、23 行：两个加数的输入框。

第 24 行：“等号”运算符按钮。

第 25 行：只读的加法结果显示框，为不可编辑的文本框。

第 28~30 行：在页面加载后的回调函数中，为等号按钮绑定单击事件。\$(document) 是 jQuery 页面加载完成后的回调函数，这个函数如果能够执行，则证明 jquery-3.1.1.js 文件引入成功。

第 32~37 行：“等号”按钮的消息响应函数，其读出 num1、num2 两个加数，求出和并赋值到结果编辑框中。

下面是 Python 的启动程序，代码如下：

```
1 import sys
2 sys.path.append('./cherry.py')
3 import cherry.py
4
5 class HelloWorld(object):
6     def __init__(self):
7         self.web_dir = '/home/osboxes/dev/wky/work/book/chp14/public/'
8
9     @cherry.py.expose
10    def index(self):
11        return 'Server start up'
12
13    @cherry.py.expose
14    def static_test(self):
15        fo = open(self.web_dir + 'static_test.html', 'r')
16        try:
17            html = fo.read()
18        finally:
19            fo.close()
20        return html
21
22 if __name__ == '__main__':
23     cherry.py.config.update({
24         'server.socket_host': '192.168.1.16',
25         'server.socket_port': 8090,
26     })
27     cherry.py.quickstart(HelloWorld(), '/', 'wky.conf')
```

程序基本没有太大的变化，只是在代码的第 15 行将打开的文件指定为 static_test.html，同时在代码的第 27 行引入编写的配置文件。

启动程序，打开浏览器，输入网址 `http://192.168.1.16:8090/static_test`，如果一切正常会出现如图 14.2 所示的界面。



图 14.2 静态资源加载示例网页显示结果

大家可以看到，在 CherryPy 框架下做一个简单的静态 Web 服务器还是非常简单的事情，我们只需要提供一些配置信息即可。

14.3 用户认证系统

如果要向外提供深度学习服务 API，第一个需要解决的问题就是用户认证问题。因为深度学习服务 API 是一个比较消耗资源的服务，我们希望只提供给授权的用户来访问。所以，需要对用户进行认证，只向合法用户提供服务。

用户认证基本有三种方式：第一种方式就是在进入网页之前，将用户强制跳转到登录页面，由用户输入用户名和密码，服务器端进行验证，认为是合法用户后就允许访问，否则拒绝访问；第二种方式是采用 RFC2617 中的基本认证方法，这种方式需要 Web 服务器支持，配置某个 URL 地址为需要认证的地址，用户访问这个地址时自动弹出登录界面，用户输入用户名和密码，验证通过后才可访问该网址；第三种方式是采用 RFC2617 中的 Digest 认证方式，这种方式同样需要 Web 服务器支持，配置某个 URL 地址为需要认证的地址，用户访问这个地址时自动弹出登录界面，用户输入用户名和密码，验证通过后才可访问该网址。

在这三种方式中，第一种方式完全由应用程序自身来实现，不依赖任何 Web 服务器，是使用最多的一种实现方式。第二种方式需要服务器支持，但是用户名和密码采用明文传输，除非在内网或配合 HTTPS 来使用，否则最好不要采用，因为安全性差。第三种方式也需要 Web 服务器支持，但是用户名和密码采用加密传输机制，因此安全性较高，在实际中应用也非常多。

我们先建立 api 目录，规定这个目录下的内容必须经过认证，在目录内建立 HTML 文件 api001.html，由于我们只是测试，文件内容可以简单一些，代码如下：

```

1 <html>
2 <body>
3 接口001
4 </body>
5 </html>

```

通过配置文件来配置 CherryPy 启动 HTTP Digest 认证，而应用程序不需要做任何处理，建立新的配置文件 `app_conf.py`，代码如下：

```

1 from cherrypy.lib import auth_digest
2
3 g_users = {'join': '123456'}
4
5 app_conf = {'/': {
6     'tools.staticdir.root': '/home/osboxes/dev/wky/work/book/chp14',
7 },
8 '/public': {
9     'tools.staticdir.on': True,
10    'tools.staticdir.dir': 'public',
11 },
12 '/api001': {
13     'tools.auth_digest.on': True,
14     'tools.auth_digest.realm': '192.168.1.16',
15     'tools.auth_digest.get_ha1': auth_digest.get_ha1_dict_plain(g_users),
16     'tools.auth_digest.key': 'a565c27146791cfb'
17 }
18 }

```

第 1 行：引入 CherryPy 的 HTTP Digest 模块。

第 3 行：保存目前网站上的合法用户，采用字典格式。大家可能会奇怪，我们原来一直使用配置文件，但是在这个例子中为什么使用 Python 程序了呢？因为 `g_users` 不是一个静态变量，会随着新用户增加而增大，因此在启动时会从数据库服务器中读出相关内容，并保存到 `g_users` 中。如果写成配置文件格式，这一过程就不能自动化了，每增加一个新用户就得改一下配置文件，极易造成错误。因此在这里采用 Python 程序的形式，可以在程序开始启动时通过数据库读取程序，利用数据库中用户表数据填充 `g_users`，因此必须采用 Python 对象形式。

第 6~11 行：在上一节中，我们已经对这些内容进行了详细讲解，这里就不再进行讲解了。

第 12 行：定义需要进行 HTTP Digest 保护的 URL。

第 13 行：打开 HTTP Digest 认证。

第 14 行：规定对哪个域进行认证，在本例中对所有请求均进行认证。

第 15 行：指定保存有用户、口令信息的源。

第 16 行：指定 key 值供服务器和浏览器进行相互认证。

下面来看启动程序，只需要将上述配置文件引入即可，在应用程序逻辑中，完全无须做任何修改，这就是 HTTP Digest 的一大好处，代码如下：

```

1 import sys
2 sys.path.append('./cherrypy')
3 import cherrypy
4 import app_conf as conf
5
6 class HelloWorld(object):
7     def __init__(self):
8         self.web_dir = '/home/osboxes/dev/wky/work/book/chp14/'
9
10    @cherrypy.expose
11    def index(self):
12        return 'Server start up'

```

```

13
14 @cherry.py.expose
15 def api001(self):
16     print('in api001???')
17     fo = open(self.web_dir + 'api/api001.html', 'r')
18     try:
19         html = fo.read()
20     finally:
21         fo.close()
22     return html
23
24 if __name__ == '__main__':
25     print(conf.app_conf)
26     cherry.py.config.update({
27         'server.socket_host': '192.168.1.16',
28         'server.socket_port': 8090,
29     })
30     cherry.py.quickstart>HelloWorld(), '/', conf.app_conf)

```

第 4 行：引入刚刚编写的配置变量。

显示 api001.html 网页内容的代码与 14.2.2 节中的代码非常类似，在这里就不再重复叙述了。

第 30 行：将第三个参数从配置文件名改为我们在此前编写的 app_conf 对象。

启动 Web 服务器，打开浏览器并访问 <http://192.168.1.16:8090/api001>，如果是新打开的浏览器，此前没有打开过这个 URL，则会出现如图 14.3 所示的页面。

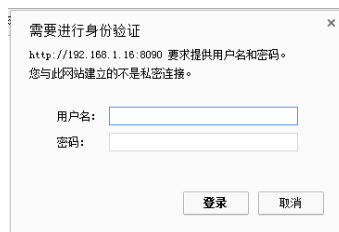


图 14.3 系统 Digest 认证输入框

当我们正确输入 join 密码为 123456 时，才可以看到 api001.html 文件的内容，如果再访问这个页面，就不用输入用户名和密码了。如果关闭了这个浏览器，打开了新的浏览器，再访问上述 URL，那么就需要重新输入用户名和密码了。

从这个例子可以看出，利用 HTTP Digest 认证，不需要我们做额外的工作就可以实现对资源的保护，还是非常方便的。

同时，如果我们想要在 HTTP Digest 认证后将其加入 Session 中，可以直接修改 `cherry.py/lib/auth_digest.py` 中的 `auth_digest` 方法，在认证成功的代码处加入我们的代码逻辑。这也是为什么采用 Git 源码而非 `pip install` 形式安装 CherryPy 的原因。因为在源码方式下，我们可以通过修改源码来满足特殊要求，而如果我们采用 `pip install` 方式安装 CherryPy 系统，过程虽然简单，但是无法获取和修改源码，因此还是源码方式更方便一些。

14.4 AJAX 请求详解

在上一节中，我们都是以网页为例来讲述 Web 服务器的开发技术。但是实际上，我们

的接口服务器接收的更多的是 AJAX 请求，并以同步或异步形式返回响应。所以本节将讲述怎样实现一个基本的 RESTful AJAX 请求处理服务器。

在这一节里，我们需要处理以下内容：创建一个基本的 RESTful 服务架构；URL 中带有参数；返回值采用 JSON 格式；大量数据采用 POST 请求，不仅需要提供服务器端的接口程序，还需要提供一个客户端调用的示例。先以浏览器为客户端，完成所有服务器端接口程序后再讲解客户端实现方式。

下面先来看 RESTful 架构，对于像我们这种需要向外界提供数据内容或计算服务的 Web 服务器而言，当前主流的方法就是采用 RESTful 架构。

RESTful 架构是开发接口服务的一系列准则，主要包括以下四个方面。

（1）严格遵守 HTTP，利用 POST 方法生成资源，例如插入数据；利用 PUT 方法改变数据内容；利用 Delete 方法删除数据；GET 方法只用于获取数据内容。

（2）采用无状态方式进行操作：为了提高并发能力和可扩展性，每个请求和响应都带有完整的信息，不需要缓存任何信息，均可以处理请求。

（3）采用目录式 URI 格式。

（4）采用 JSON 等数据格式进行传输。

14.4.1 添加数据

下面先用 HTTP POST 请求来创建数据，RESTful 网页部分的实现代码如下：

```

1 <!DOCTYPE HTML>
2 <html lang="en">
3 <head>
4 <meta charset="utf-8">
5 <title>测试</title>
6 <meta name="viewport" content="width=device-width, initial-scale=1.0">
7 <meta name="description" content="">
8 <meta name="author" content="">
9 <script type="text/javascript" src="public/js/jquery-3.1.1.js"></script>
10 <link rel="stylesheet" type="text/css" href="public/css/wky.css">
11 <!--bootstrap-->
12 <script type="text/javascript" src="public/js/bootstrap.min.js"></script>
13 <link rel="stylesheet" type="text/css" href="public/js/bootstrap.min.css">
14 </head>
15 <body>
16 <span>REST POST : 添加资源</span><br />
17 <input type="text" id="user_name" value="" placeholder="用户名" /><br />
18 <input type="text" id="email" value="" placeholder="邮件" /><br />
19 <input type="button" id="saveBtn" value="保存" />
20
21 <script type="text/javascript">
22 $(document).ready(function() {
23     $("#saveBtn").bind("click", save);
24 });
25
26 function save() {
27     var reqUrl = "http://192.168.1.16:8090/rest01_do"
28     var user_name = $("#user_name").val();
29     var email = $("#email").val();
30     var data = new Object();
31     data.user_name = user_name;
32     data.email = email;
33     $.ajax({
34         url: reqUrl,
35         type: 'POST',

```

```

36     dataType: 'json',
37     data: {
38         json_str: JSON.stringify(data)
39     },
40     success: onSaveOk,
41     error: onSaveError
42 });
43 }
44
45 function onSaveOk(json) {
46     alert("保存用户成功：" + json.user_id + "！");
47 }
48 function onSaveError(msg) {
49     alert("保存用户失败：" + JSON.stringify(msg) + "！");
50 }
51 </script>
52
53 </body>
54 </html>

```

在这个文件中，对于我们当前的任务而言，最重要的是第 26 行定义的 save 函数。

第 27 行：定义 AJAX 请求的 URL。

第 29、30 行：读取屏幕中相应控件的内容，必要时可以在此处进行数据合法性检查，由于此处是示例，所以数据合法性检查就省略了。

第 31、32 行：定义 JavaScript 对象保存所有参数。

第 33~42 行：利用 jQuery 的 AJAX 的 POST 请求，发送 AJAX 请求，在这里将 type 指定为 POST、dataType 指定为 JSON 格式，data 中将保存数据的 Object 序列化为字符串，作为 POST 数据，并且指定成功和失败的消息响应函数。

第 45~47 行：保存成功的消息响应函数，只响应服务器端返回的 user_id 值。

第 48~50 行：保存失败的消息响应函数，显示出错信息。

上面的 save 函数其实就是 REST API 的 AJAX 请求示例，后面还会给出 Python 代码实现的客户端，完成同样的功能。

下面来看服务器的实现代码，如下：

```

1 import sys
2 import json
3 sys.path.append('./cherry.py')
4 import cherry.py
5 import app_conf as conf
6
7 class HelloWorld(object):
8     def __init__(self):
9         self.web_dir = '/home/osboxes/dev/wky/work/book/chp14/'
10
11     @cherry.py.expose
12     def index(self):
13         return 'Server start up'
14
15     @cherry.py.expose
16     def rest01(self):
17         html = self.read_html('public/rest01.html')
18         return html
19
20     @cherry.py.expose
21     @cherry.py.tools.json_out()
22     def rest01_do(self, json_str):
23         print('*****\r\n REST POST example \r\n*****')
24         json_obj = json.loads(json_str)
25         print('保存用户：userName=%s; email=%s' % (json_obj['user_name'], \
26             json_obj['email']))
27         print('*****\r\n REST POST example \r\n*****')
28         resp = {'status': 'Ok'}
29         resp['user_id'] = 8008
30         return resp
31

```



```

32     @cherry.py.expose
33     def api001(self):
34         html = self.read_html('api/api001.html')
35         return html
36
37     def read_html(self, file):
38         fo = open(self.web_dir + file, 'r')
39         try:
40             html = fo.read()
41         finally:
42             fo.close()
43         return html
44
45 if __name__ == '__main__':
46     print(conf.app_conf)
47     cherry.py.config.update({
48         'server.socket_host': '192.168.1.16',
49         'server.socket_port': 8090,
50     })
51     cherry.py.quickstart(HelloWorld(), '/', conf.app_conf)

```

此段程序中很大一部分代码与之前讲解的代码是相同的，这里就不再重复讲解了，重点讲解 POST 请求处理部分。

第 2 行：由于要通过 JSON 格式返回响应结果，并将请求体解析为 JSON 变量，所以需要引入 json 库。

第 15~18 行：显示上一步定义的网页。

第 21 行：定义消息响应函数，返回的值为 JSON 格式。

第 22 行：具体 REST API 定义函数，其第二个参数为 AJAX POST 请求传递过来的参数。

第 24 行：先将请求体解析为 JSON 变量。

第 25、26 行：由于我们还没讲数据库操作部分，所以这里并没有真正保存，只是在后台打印出用户名和邮件。

第 28 行：生成字典类型对象 resp，用于保存响应中的内容。

第 29 行：将插入用户表中产生的主键 user_id 返回给客户端。

其余部分的代码与前面例子的大体相同，唯一有所区别的就是将读取 HTML 文件内容的代码封装为 self.read_html 函数，读者应该可以明白余下代码的含义。运行上面的程序，结果如下：

```

*****
REST POST example
*****
保存用户 : userName=yant; email=yant@wky.com
*****
REST POST example
*****

```

这表明在接口中已经可以成功获取 POST 过来的参数，余下的任务就是向数据库插入记录，记录并向客户端返回新记录主键的工作了。

14.4.2 修改数据

根据 RESTful 理论，我们需要修改某个内容时应调用 HTTP 的 PUT 请求，先在请求体中添加相应内容，然后后台接口程序解析这些数据，完成相应的数据更新操作，最后将数据更新结果返回给客户端。

在客户端的网页还是沿用 POST 请求的网络，只是在其中添加了一个 hidden 控件和一个“修改”按钮，以及点击事件消息响应函数，代码如下：

```

20 <input type="hidden" id="user_id" value="8008" />
21 <input type="button" id="updateBtn" value="修改" /><br />
24 $(document).ready(function() {
25     $("#saveBtn").bind("click", save);
26     $("#updateBtn").bind("click", update);
27 });

55 function update() {
56     var reqUrl = "http://192.168.1.16:8090/rest01_update"
57     var user_id = $("#user_id").val();
58     var user_name = $("#user_name").val();
59     var email = $("#email").val();
60     var data = new Object();
61     data.user_id = user_id;
62     data.user_name = user_name;
63     data.email = email;
64     $.ajax({
65         url: reqUrl,
66         type: 'PUT',
67         dataType: 'json',
68         data: {
69             json_str: JSON.stringify(data)
70         },
71         success: onUpdateOk,
72         error: onUpdateError
73     });
74 }
75 function onUpdateOk(json) {
76     if ("Ok" == json.status) {
77         alert("更新信息成功！");
78     } else {
79         alert("更新信息失败！");
80     }
81 }
82 function onUpdateError(msg) {
83     alert("更新失败：" + JSON.stringify(msg) + "！");
84 }

```

第 20 行：定义一个隐藏编辑框，保存 user_id，在更新时提交给服务器，表明需要删除哪个用户。

第 21 行：向页面中添加“修改”按钮。

第 26 行：向“修改”按钮添加单击事件消息响应函数。

第 55 行：定义“修改”按钮消息响应函数。

第 56 行：定义 PUT 请求 URL。

第 57~59 行：从页面中获取用户基本信息，可以在此处做数据合法性检查，本处省略了这部分代码。

第 60~63 行：生成 JavaScript 的 Object 对象，保存用户的基本信息。

第 64~73 行：向服务器端发送 PUT 请求，此时 dataType 的类型为 PUT，其他与 POST 请求基本相同。

第 75~81 行：修改成功的消息响应函数。

第 82~84 行：修改失败的消息响应函数。

下面来看服务器的实现，代码如下：

```

32 @cherry.py.expose
33 @cherry.py.tools.json_out()
34 def rest01_update(self, json_str):
35     print('*****\r\n REST PUT example \r\n*****')
36     json_obj = json.loads(json_str)
37     print('修改用户：user_id=%d:user_name=%s; email=%s' % \
38         (int(json_obj['user_id']), json_obj['user_name'], \
39          json_obj['email']))
40     print('*****\r\n REST PUT example \r\n*****')
41     resp = {'status': 'Ok'}
42     return resp

```

第 33 行：指定返回给客户端的响应为 JSON 格式。

第 34 行：定义修改数据 PUT 请求响应函数，第二个参数为客户端传过来的 JSON 字符串，包含所有参数信息。

第 36 行：将第二个参数解析为 JSON 变量。

第 37~39 行：在实际应用中，我们会根据参数内容做数据更新操作，但是在此处所做的仅仅是将信息打印到界面中。

第 41 行：生成字典对象，保存响应中的信息。

第 42 行：以 JSON 格式返回响应内容。

14.4.3 删除数据

在进行这部分操作之前，我们先来看一下 CherryPy 的实现。在默认状态下，CherryPy 认为只有 POST、PUT、PATCH 请求会有请求体，如果发送 DELETE 请求的请求体，CherryPy 会将其自动忽略掉，这显然不是我们想要的结果，所以需要改变 CherryPy 的默认行为，允许 DELETE 请求具有请求体。读者在这里就可以看到采用源码方式集成，而不是安装到系统中的好处了！我们可以任意定制 CherryPy 来满足特定需求。

打开 `cherrypy/_cprequest.py` 文件，定位在第 315 行：

```
313     and the result placed into request.params or request.body."""
314
315     methods_with_bodies = ('POST', 'PUT', 'PATCH')
316     """
```

由上面的代码可以看出，在默认情况下，CherryPy 只允许 POST、PUT、PATCH 方法有请求体，我们需要 DELETE 请求也有请求体，因此需要将上面的代码修改为：

```
313     and the result placed into request.params or request.body."""
314
315     methods_with_bodies = ('POST', 'PUT', 'PATCH', 'DELETE')
316     """
```

在对 CherryPy 进行了一个小的修改之后，再来看怎样实现 DELETE 请求。

页面中的内容如下：

```
20 <input type="hidden" id="user_id" value="8008" />
21 <input type="button" id="updateBtn" value="修改" /><br />
22 <input type="button" id="deleteBtn" value="删除" /><br />
```

```
25 $(document).ready(function() {
26     $("#saveBtn").bind("click", save);
27     $("#updateBtn").bind("click", update);
28     $("#deleteBtn").bind("click", doDelete);
29 });
```

```
89 function doDelete() {
90     var reqUrl = "http://192.168.1.16:8090/rest01_delete"
91     var user_id = $("#user_id").val();
92     var data = new Object();
93     data.user_id = user_id;
94     alert(JSON.stringify(data));
95     $.ajax({
96         url: reqUrl,
97         type: 'DELETE',
98         dataType: 'json',
```

```

99     data: {
100         json_str: JSON.stringify(data)
101     },
102     success: onDeleteOk,
103     error: onDeleteError
104 });
105 }
106 function onDeleteOk(json) {
107     if ("Ok" == json.status) {
108         alert("删除信息成功!");
109     } else {
110         alert("删除信息失败!");
111     }
112 }
113 function onDeleteError(msg) {
114     alert("删除失败：" + JSON.stringify(msg) + " !");
115 }

```

第 22 行：添加“删除”按钮。

第 28 行：添加“删除”按钮单击消息响应函数 doDelete。

第 89 行：定义 doDelete 函数。

第 90 行：指定删除数据接口 URL。

第 91 行：取出需要删除用户的 user_id，因为是删除数据，所以只需要主键值就可以了。在实际应用中，需要对主键值进行合法性验证。

第 92、93 行：调用 JavaScript 对象，并保存要删除数据的信息。

第 95~104 行：向服务器发送 DELETE 请求，type 的值为 DELETE。

第 106~112 行：定义删除成功消息响应函数。

第 113~115 行：定义删除失败消息响应函数。

14.4.4 REST 服务实现

GET 请求与增加、修改、删除操作不同，是从服务器端获取数据。为了向服务器端表明需要获取哪些数据，GET 请求有两种方式向服务器端传递参数：第一种是请求目录形式，如 `http://server_ip/getData/userId/500/username/王`；第二种是通过 QueryString，如 `http://server_ip/getData?user_id=500&user_name=王`。当然，这两种方式也可以结合在一起使用。

1. GET 请求

下面将对 CherryPy 框架进行修改，使我们可以比较方便地同时支持这两种方式。同时，我们也将修改我们的主程序，使其成为完全符合 REST 风格的程序。

下面进行对 CherryPy 框架的修改，在 `cherrypy/_cpdispatch.py` 文件中，代码如下：

```

58     def __call__(self):
59         try:
60             params = {}
61             params['args'] = self.args
62             params['kwargs'] = self.kwargs
63             self.args = ()
64             return self.callable(params)
65             #return self.callable(*self.args, **self.kwargs)
66         except TypeError:
67             x = sys.exc_info()[1]
68             try:
69                 test_callable_spec(self.callable, self.args, self.kwargs)

```

```

70         except cherrypy.HTTPError:
71             raise sys.exc_info()[1]
72     except:
73         raise x
74     raise

```

这段代码表明 CherryPy 找到请求对应的方法之后，将调用该方法，并将参数传递给该方法。在原有的实现中，直接将一个元组和字典作为参数传给目标函数，如代码第 65 行所示。但是这种方法非常不灵活，需要目标函数显式接收 `QueryString` 中的参数。另外，很难处理路径形式的参数，因此我们采用了自己的实现，将元组和字典统一放在一个字典中，并作为可选参数传给目标函数，这样就可以自由地使用路径中的参数和 `QueryString` 中的参数了，并且可以采用符合 REST 风格的 URL。

改造完 CherryPy 框架之后，再来看怎样实现 REST 风格的 URL。我们知道，REST 风格的 URL 通常采用 GET /wky/bes/user 的格式，而这种格式放在前面的程序中是有问题的。所以我们需要使用 `MethodDispatcher` 和 `cherrypy.engine` 来启动服务器。

首先来看配置信息的修改：

```

1 import sys
2 sys.path.append('./cherrypy')
3 import cherrypy
4 from cherrypy.lib import auth_digest
5
6 g_users = {'join': '123456'}
7
8 app_conf = {'/': {
9     'tools.staticdir.root': '/home/osboxes/dev/wky/work/book/chp14',
10    'request.dispatch': cherrypy.dispatch.MethodDispatcher(),
11 },
12 '/web/public': {
13     'tools.staticdir.on': True,
14     'tools.staticdir.dir': 'public',
15 },
16 '/api001': {
17     'tools.auth_digest.on': True,
18     'tools.auth_digest.realm': '192.168.1.16',
19     'tools.auth_digest.get_ha1': auth_digest.get_ha1_dict_plain(g_users),
20     'tools.auth_digest.key': 'a565c27146791cfb'
21 }
22 }

```

第 10 行：配置 `request.dispatch` 属性为 `MethodDispatcher` 类，这样就可以使用 REST 风格的 URL 了。

第 12 行：因为我们将要使用 `/web/pages` 这个 URL 来显示普通 HTML 页面，因此根目录就是 `/web`，所以需要配置 CSS、JS 等资源全在这个子 URL 下面。

第 16~21 行：HTTP Digest 认证相关代码。

接着看服务器启动代码，由于我们现在的业务逻辑复杂了，所以不能使用 `quickstart` 来启动 Web 服务器了，启动代码如下：

```

1 import sys
2 sys.path.append('./cherrypy')
3 import json
4 import cherrypy
5 from page_controller import PageController
6 import app_conf as conf
7
8 class MainController(object):
9     exposed=True
10     def __init__(self):
11         self.web_dir = '/home/osboxes/dev/wky/work/book/chp14/'
12

```

```

13 def get_users(self, params):
14     print(params)
15     resp = {'status': 'Ok'}
16     resp['datas'] = []
17     uo = {'user_id': 1001, 'user_name': 'u1001'}
18     resp['datas'].append(uo)
19     uo = {'user_id': 1002, 'user_name': 'u1002'}
20     resp['datas'].append(uo)
21     return resp
22
23 def save_user(self, params):
24     print('save_user:%s' % params['kwargs']['json_obj'])
25     resp = {'status': 'Ok'}
26     resp['user_id'] = 8008
27     resp['user_name'] = '中文名'
28     return resp
29
30 def update_user(self, params):
31     print('update_user:%s' % params['kwargs']['json_obj'])
32     json_obj = params['kwargs']['json_obj']
33     resp = {'status': 'Ok'}
34     resp['user_id'] = json_obj['user_id']
35     return resp
36
37 def delete_user(self, params):
38     print('delete_user:%s' % params['kwargs']['json_obj'])
39     json_obj = params['kwargs']['json_obj']
40     resp = {'status': 'Ok'}
41     resp['user_id'] = json_obj['user_id']
42     return resp
43
44
45 def unknown(self, params):
46     resp = {'status': 'Error', 'msg': '没有命令'}
47     return resp
48
49 @cherry.py.tools.json_out()
50 def GET(self, params={'cmd': 'unknown'}):
51     return self.http_method(params)
52
53 @cherry.py.tools.json_out()
54 def POST(self, params={'cmd': 'unknown'}):
55     json_obj = json.loads(params['kwargs']['json_str'])
56     del params['kwargs']['json_str']
57     params['kwargs']['json_obj'] = json_obj
58     params['kwargs']['cmd'] = json_obj['cmd']
59     return self.http_method(params)
60
61 @cherry.py.tools.json_out()
62 def PUT(self, params={'cmd': 'unknown'}):
63     json_obj = json.loads(params['kwargs']['json_str'])
64     del params['kwargs']['json_str']
65     params['kwargs']['json_obj'] = json_obj
66     params['kwargs']['cmd'] = json_obj['cmd']
67     return self.http_method(params)
68
69 @cherry.py.tools.json_out()
70 def DELETE(self, params={'cmd': 'unknown'}):
71     json_obj = json.loads(params['kwargs']['json_str'])
72     del params['kwargs']['json_str']
73     params['kwargs']['json_obj'] = json_obj
74     params['kwargs']['cmd'] = json_obj['cmd']
75     return self.http_method(params)
76
77 def http_method(self, params):
78     cmd = params['kwargs'].get('cmd', 'unknown')
79     func = getattr(self, cmd)
80     return func(params)
81
82 def read_html(self, file):
83     fo = open(self.web_dir + file, 'r')
84     try:
85         html = fo.read()
86     finally:
87         fo.close()
88     return html

```

```

89
90 if __name__ == '__main__':
91     cherrypy.config.update({
92         'server.socket_host': '192.168.1.16',
93         'server.socket_port': 8090,
94     })
95     pc = PageController()
96     mc = MainController()
97     cherrypy.tree.mount(pc, '/web/pages', conf.app_conf)
98     cherrypy.tree.mount(mc, '/wky/test/c009', conf.app_conf)
99     cherrypy.engine.start()
100    cherrypy.engine.block()

```

下面对 Web 服务器启动代码进行解析。

第 91~93 行：监听在物理 IP 地址和 8090 端口。

第 95 行：页面显示用控制器，主要用于显示 HTML 页面。

第 96 行：REST 风格服务控制器，用于实现接口。

第 97 行：所有/web/pages 请求，根据 QueryString 中的参数 cmd=show_***显示对应的 HTML 文件内容。

第 98 行：所有/wky/test/c009 为接口地址，可以接收 POST(添加)、PUT(修改)、DELETE(删除)、GET(查询)请求，通过消息体或 QueryString 中的 cmd=***来决定调用哪个函数，并返回 JSON 字符串。

第 99、100 行：启动 Web 服务器。

下面我们来看 GET 请求处理，服务器代码如下。

第 9 行：如果我们希望使用 REST 风格的 URL，必须指定这个风格。

第 49 行：定义这个函数将返回 json 格式的数据。

第 50 行：如果客户端发过来的请求为 GET /wky/test/c009?cmd=get_users HTTP/1.1，则自动调用这个函数，该函数中的 params 为可选参数，是字典类型，具有元组 args 和字典 kwargs 两个元素，其中 kwargs 中保存 QueryString 和消息体中的参数，而 args 元组保存 URL 路径中的参数。

第 51 行：调用 HTTP 处理函数 http_method。

http_method 方法实现如下。

第 77 行：定义 http_method 函数，参数为字典类型，具有元组 args 和字典 kwargs 两个元素，其中 kwargs 中保存 QueryString 和消息体中的参数，而 args 元组保存 URL 路径中的参数。

第 78 行：从参数中取出 cmd 参数，如果没有就设置为 unknown。

第 79 行：查找 cmd 同名函数。

第 80 行：调用该函数，这里用的是 Python 的反射机制，如果读者对这部分内容不熟悉，可以在网上查询一下相关概念。

在这里，由于我们使用 cmd=get_users，所以会执行 get_users 方法。

第 13 行：定义 get_users 方法，参数 params 为字典类型，具有元组 args 和字典 kwargs 两个元素，其中 kwargs 中保存 QueryString 和消息体中的参数，而 args 元组保存 URL 路径中的参数。

第 15 行：定义响应为一个字典变量。

第 16 行：用户列表用 `datas` 来表示。

第 17~20 行：`datas` 列表中每个元素均为一个字典。

第 21 行：返回 `resp` 响应字典。

打开浏览器，在地址栏中输入 `http://192.168.1.16:8090/wky/test/c009?cmd=get_users`，会显示如图 14.4 所示的内容。



图 14.4 GET 请求结果

由图 14.4 可以看出，REST GET 接口是可以正确地向外界提供服务的。

2. POST 请求

在 REST 体系下，POST 请求用于创建数据，在本例中其用于创建新用户，用户的数据将在请求体中，采用 JSON 格式。服务器端接口实现如下。

第 53 行：定义该函数的输出为 JSON 格式。

第 54 行：定义 POST 方法的响应函数，参数 `params` 为字典类型，具有元组 `args` 和字典 `kwargs` 两个元素，其中 `kwargs` 中保存 `QueryString` 和消息体中的参数，而 `args` 元组保存 URL 路径中的参数。

第 55 行：将消息体中传递来的内容从字符串形式解析为 JSON 对象 `json_obj`。

第 56 行：从参数中将原来的 JSON 字符串删除。

第 57 行：将新创建的 JSON 对象加入到参数中。

第 58 行：将消息体中 `cmd` 域加入到参数中，为调用 `http_method` 方法做好准备。

第 59 行：调用 `http_method` 方法，最后调用到合适的方法。

在 `http_method` 中（代码解析见 GET 请求部分），将根据 `cmd` 内容调用相应的方法。在本例中将调用 `save_user` 方法。

第 23 行：定义 `save_user` 方法。

第 24 行：参数中的 `json_obj` 为消息体中的具体内容，本例中为要保存用户的基本信息，实际应用中会将这些信息保存到数据库中，这里仅打印出其内容。

第 25 行：定义 `resp` 字典变量作为响应对象。

第 26 行：在添加操作中，一般将刚添加记录的主键返回给调用者。

第 28 行：返回 `resp`，会经过本类的 POST 方法将其转换为 JSON 格式，并发送给客户端。

下面再来看客户端的实现方式，我们还采用 `public/rest01.html`，只需要将 AJAX 请求 URL 改为当前 REST URL，并在消息体中指定 `cmd` 内容即可，代码如下：


```

31 function save() {
32     var reqUrl = "http://192.168.1.16:8090/wky/test/c009"
33     var user_name = $("#user_name").val();
34     var email = $("#email").val();
35     var data = new Object();
36     data.cmd = 'save_user'
37     data.user_name = user_name;
38     data.email = email;
39     $.ajax({
40         url: reqUrl,
41         type: 'POST',
42         dataType: 'json',
43         data: {
44             json_str: JSON.stringify(data)
45         },
46         success: onSaveOk,
47         error: onSaveError
48     });
49 }
50
51 function onSaveOk(json) {
52     alert("保存用户成功：" + json.user_id + "！");
53 }
54 function onSaveError(msg) {
55     alert("保存用户失败：" + JSON.stringify(msg) + "！");
56 }

```

第 32 行：将 AJAX 请求的 URL 改为目前 REST 请求的 URL。

第 36 行：指定 cmd 参数，这个参数会决定调用哪个方法。

其余部分代码与 14.4.1 节中的代码基本一致，这里就不再赘述了。

3. PUT 请求

在标准 REST 体系中，更新数据的操作采用 PUT 请求，将需要更新的数据作为消息体传递给服务器端，由服务器端完成相应的修改操作。

我们先来看服务器端的实现。

第 61 行：定义本函数返回值为 JSON 格式。

第 62 行：定义 PUT 方法，用于处理 HTTP PUT 请求，参数 params 为字典类型，具有元组 args 和字典 kwargs 两个元素，其中 kwargs 中保存 QueryString 和消息体中的参数，而 args 元组保存 URL 路径中的参数。

第 63 行：将消息体中的参数 json_str 取出来，并将其转换为 JSON 对象 json_obj，其内容为需要更新的数据。

第 64 行：从 params 参数中删除 json_str。

第 65 行：将新创建的 json_obj 保存到 params 中。

第 66 行：从 json_obj 中取出 cmd 参数，并保存到 params 中，为调用 http_method 方法做好准备。

第 67 行：调用 http_method 方法，找到最终需要调用的方法，并调用该方法。在本例中，最终调用的函数为 update_user。

具体数据修改操作将在 update_user 中实现。

第 30 行：定义 update_user 函数，参数 params 为字典类型，具有元组 args 和字典 kwargs 两个元素，其中 kwargs 中保存 QueryString 和消息体中的参数，而 args 元组保存 URL 路径中的参数。

第 32 行：从参数中取出 `json_obj`，其内容为需要更新的数据内容。

第 33 行：定义字典 `resp`，保存返回内容。

第 35 行：返回 `resp`，在 PUT 函数中，将其转换为 JSON 格式。

下面来看网页客户端实现，代码如下：

```
58 function update() {
59     var reqUrl = "http://192.168.1.16:8090/wky/test/c009"
60     var user_id = $("#user_id").val();
61     var user_name = $("#user_name").val();
62     var email = $("#email").val();
63     var data = new Object();
64     data.cmd = "update_user";
65     data.user_id = user_id;
66     data.user_name = user_name;
67     data.email = email;
68     $.ajax({
69         url: reqUrl,
70         type: 'PUT',
71         dataType: 'json',
72         data: {
73             json_str: JSON.stringify(data)
74         },
75         success: onUpdateOk,
76         error: onUpdateError
77     });
78 }
79 function onUpdateOk(json) {
80     if ("Ok" == json.status) {
81         alert("更新信息成功!");
82     } else {
83         alert("更新信息失败!");
84     }
85 }
86 function onUpdateError(msg) {
87     alert("更新失败：" + JSON.stringify(msg) + " !");
88 }
```

代码主体与 14.4.2 节中的代码相同，只有以下两点区别：

第 59 行：调用新的 REST 风格 URL。

第 64 行：指定 `cmd` 参数，其将决定最终调用哪个方法。

4. DELETE 请求

在标准 REST 体系中，DELETE 请求用于删除数据，要删除的数据放在消息体中，采用 JSON 格式传输。

下面来看 DELETE 请求在服务器端的实现。

第 69 行：定义函数的返回值为 JSON 格式。

第 70 行：定义 DELETE 方法，用于处理 HTTP DELETE 请求，参数 `params` 为字典类型，具有元组 `args` 和字典 `kwargs` 两个元素，其中 `kwargs` 中保存 `QueryString` 和消息体中的参数，而 `args` 元组保存 URL 路径中的参数。

第 71 行：从消息体中取出 `json_str` 字符串，并将其转化为 JSON 对象，其内容为要删除的数据内容。

第 72 行：从 `params` 中删除 `json_str`。

第 73 行：将新创建的 `json_obj` 保存到 `params` 中。

第 74 行：将消息体中的 `cmd` 参数保存到 `params` 中，为调用 `http_method` 做好准备。

第 75 行：调用 `http_method` 方法，其将调用 `cmd` 参数规定的同名方法，在本例中为

delete_user 方法。

在 delete_user 方法中，完成具体的数据删除操作。

第 37 行：定义 delete_user 方法，参数 params 为字典类型，具有元组 args 和字典 kwargs 两个元素，其中 kwargs 中保存 QueryString 和信息体中的参数，而 args 元组保存 URL 路径中的参数。

第 39 行：从 params 中取出要删除的数据 json_obj。

第 40 行：定义 resp 字典保存需要返回给客户端的信息。

第 42 行：返回 resp，在 DELETE 方法中会将其转换为 JSON 格式返回给客户端。

下面来看网页部分的实现，代码如下：

```

91 function doDelete() {
92     var reqUrl = "http://192.168.1.16:8090/wky/test/c009"
93     var user_id = $("#user_id").val();
94     var data = new Object();
95     data.cmd = "delete_user"
96     data.user_id = user_id;
97     $.ajax({
98         url: reqUrl,
99         type: 'DELETE',
100        dataType: 'json',
101        data: {
102            json_str: JSON.stringify(data)
103        },
104        success: onDeleteOk,
105        error: onDeleteError
106    });
107 }
108 function onDeleteOk(json) {
109     if ("Ok" == json.status) {
110         alert("删除信息成功!");
111     } else {
112         alert("删除信息失败!");
113     }
114 }
115 function onDeleteError(msg) {
116     alert("删除失败：" + JSON.stringify(msg) + " !");
117 }

```

程序主体与 14.4.3 节中的基本相同，主要有以下两点区别：

第 92 行：将调用 URL 改换为 REST 风格的 URL。

第 95 行：指定 cmd 参数，其将决定最终调用的方法。

5. 网页显示

我们已经介绍了 REST 服务的实现技术，但是还遗留了一个重要的问题——网页显示。

我们通过定义 PageController 来显示网络，代码如下：

```

1 import sys
2 sys.path.append('./cherrypy')
3 import cherrypy
4
5 class PageController(object):
6     exposed=True
7     def __init__(self):
8         self.web_dir = '/home/osboxes/dev/wky/work/book/chp14/'
9
10    def show_rest01(self, params):
11        return self.read_html('public/rest01.html')
12
13    def GET(self, params={}):
14        cmd = params['kwargs']['cmd']
15        func = getattr(self, cmd)

```

```

16         return func(params)
17
18     def read_html(self, file):
19         fo = open(self.web_dir + file, 'r')
20         try:
21             html = fo.read()
22         finally:
23             fo.close()
24         return html

```

第 6 行：由于我们是 REST 风格的 URL，所以必须设置 `exposed = True`。

第 7、8 行：定义构造函数，指定 Web 页面的根目录。

第 10 行：定义 `show_rest01` 方法，具体显示 `rest01.html` 页面。

第 11 行：调用本类的 `read_html` 方法，从文件中读出内容，并返回给客户端。

第 13 行：定义 GET 方法，处理 HTTP GET 请求。因为我们通过配置文件已经设定，所以页面显示 URL 格式为 `http://server_ip/web/pages?cmd=show_rest01`，其中 `show_rest01` 为本类方法，显示某个具体的网页。

第 14 行：取出 `cmd` 参数。

第 15 行：在本类中查找与 `cmd` 参数同名的方法。

第 16 行：调用该方法。

14.5 数据持久化技术

在前面各节我们都没有进行数据库操作，在这一节中我们将向读者介绍数据库操作技术。我们将采用 MySQL 数据库，数据库操作技术采用 PyMySQL。在这一节中，不仅向读者介绍数据库的增加、修改、删除和查询操作，还将介绍事务、数据库连接池的相关内容。

14.5.1 环境搭建

需要确保在机器上安装了 MySQL，如果没有安装，则直接运行 `sudo apt-get install mysql` 进行安装。先创建演示数据库，以 `root` 身份登录 MySQL 并运行以下命令：

```

mysql> create database DlbDb default character set utf8 collate utf8_general_ci;
mysql> grant all privileges on DlbDb.* to dlb@'%' identified by 'dlb123';
mysql> grant all privileges on DlbDb.* to dlb@'localhost' identified by 'dlb123'
;

```

退出 MySQL，然后以用户名 `dlb` 和密码 `dlb123` 登录 `DlbDb` 数据库：

```
mysql -udlb -pdb123 DlbDb
```

再创建数据库表，代码如下：

```

1 create table t_user(
2     user_id bigint primary key auto_increment,
3     user_name varchar(200),
4     email varchar(200),
5     login_name varchar(20),
6     login_pwd varchar(20),
7     salary double,
8     create_date datetime
9 );
10
11 create table t_acct(
12     acct_id bigint primary key auto_increment,
13     acct_name varchar(200)
14 );
15
16 create table t_user_acct(
17     user_id bigint,
18     acct_id bigint
19 );

```

下面简单介绍一下数据库表的用途。

- (1) 用户表（t_user）：保存用户基本信息。
- (2) 账户表（t_acct）：保存账户基本信息。
- (3) 用户账户表（t_user_acct）：保存用户与账户间的关联关系。

在创建账户时，需要同时在账户表 and 用户账户表中添加记录，必须同时成功或失败，我们会将这一需求采用数据库事务来实现。

我们将使用 PyMySQL 来操作数据库，所以须安装这一模块：

```
22 pip install pymysql
```

14.5.2 数据库添加操作

下面向数据库中添加记录，分为两种情况来讨论，第一种情况是一次添加一条记录，第二种情况是一次添加多条记录。

下面先来看一次添加一条记录，我们将向 t_user 表中插入一条记录，并打印新加入记录的主键，代码如下：

```

1 import pymysql
2
3 conn = pymysql.connect(host='localhost', port=3306, user='dlb', \
4     passwd='dlb123', db='DLbDb', charset='utf8')
5 cursor = conn.cursor()
6 affected_rows = cursor.execute('insert into t_user(user_name, \
7     email, login_name, login_pwd, create_date) \
8     values(%s, %s, %s, %s, sysdate())', \
9     ('u1', 'u1@a.com', 'u1log', 'u123456'))
10 conn.commit()
11 cursor.close()
12 conn.close()
13 pk = cursor.lastrowid
14 print('pk=%d; affected_rows=%d' % (pk, affected_rows))

```

第 1 行：引入 pymysql 模块。

第 3 行：用指定用户名口令连接指定数据库。

第 5 行：获取数据库操作所需游标 cursor。

第 6~9 行：在游标 cursor 上调用 execute 函数执行 SQL 语句，该函数第一个参数为一

条带有参数的 SQL 语句，参数用类似 print 的格式给出。在本例中为 4 个字符型参数，第二个参数为一个元组，里面是 SQL 语句中参数的值。

第 10 行：调用 conn 的 commit 方法，将修改更新到数据表中。

第 11 行：关闭游标 cursor，数据库完成后需要关闭游标，释放资源。

第 12 行：关闭数据库连接。由于本例只是一个示例，因此我们关闭数据库连接。在实际应用中，我们会采用数据库连接池（将在本节最后进行讨论），是不需要关闭数据库连接的，只需将数据库连接放回连接池就可以了。

第 13 行：获取刚添加到数据库中记录的主键。

第 14 行：打印主键和实际插入的行数。

一次插入多条数据的代码与一次插入一条数据的代码类似，都需要 execute 函数，这是因为它会大大提高效率，因此这个方法在数据导入/导出时用的最多，代码如下：

```
1 import pymysql
2
3 conn = pymysql.connect(host='localhost', port=3306, user='dlb', \
4                        passwd='dlb123', db='DlbDb', charset='utf8')
5 cursor = conn.cursor()
6 affected_rows = cursor.executemany('insert into t_user(user_name, \
7                                   email, login_name, login_pwd, create_date) \
8                                   values(%, %, %, %, sysdate())', \
9                                   [('u1', 'u1@a.com', 'u1log', 'u123456')])
10 conn.commit()
11 cursor.close()
12 conn.close()
13 pk = cursor.lastrowid
14 print('pk=%d; affected_rows=%d' % (pk, affected_rows))
```

上面的代码与一次插入一条记录的代码类似，只有两点需要说明：

第 6~9 行：调用 cursor.executemany 函数，表明要一次插入多条记录，函数的第二个参数不再是元组，而是元组的列表，每条记录用一个元组来表示。

第 13 行：取出的是最后插入记录的主键。

14.5.3 数据库修改操作

下面以修改 user_id=1 记录的用户名为“王小明”为例，向读者演示数据库修改操作的实现方式，代码如下：

```
1 import pymysql
2
3 conn = pymysql.connect(
4     host='localhost',
5     port=3306,
6     user='dlb',
7     passwd='dlb123',
8     db='DlbDb',
9     charset='utf8'
10 )
11
12 cursor = conn.cursor()
13 sql = 'update t_user set user_name=%, email=%, salary=%s where \
14       user_id=%s'
15 params = ('王小明', 'wxm@abc.com', 15998.99, 1)
16 affected_rows = cursor.execute(sql, params)
17 conn.commit()
18 cursor.close()
```

```
19 conn.close()
20 print('affected_rows=%d' % affected_rows)
```

第 1 行：引入 pymysql 库。

第 3~10 行：建立数据库连接。

第 12 行：从数据库连接 conn 中获取操作数据库所需的游标 cursor。

第 13 行：定义更新数据库的 SQL 语句，语句中的参数用%s 作为占位符，整数、实数类型也用%s 作为占位符，而不使用%d、%f。

第 15 行：用元组提供参数的数值。

第 16 行：数据库更新 SQL 语句。

第 17 行：将数据库更新写入数据中。

第 18 行：关闭游标 cursor。

第 19 行：关闭数据库连接，如果使用数据库连接池，则是将数据库连接放回到数据库连接池中。

第 20 行：打印更新的行数。注意：如果要更新的内容与数据库中的内容完全一致，则更新行数为 0，不能简单地通过这个数判断是否更新成功。

14.5.4 数据库删除操作

下面来看数据库删除操作，在这个例子中，将要删除数据库中 user_id>=2 的记录，代码如下：

```
1 import pymysql
2
3 conn = pymysql.connect(
4     host='localhost',
5     port=3306,
6     user='dlb',
7     passwd='dlb123',
8     db='DlbDb',
9     charset='utf8'
10 )
11
12 cursor = conn.cursor()
13 sql = 'delete from t_user where user_id>=2'
14 params = (2)
15 affected_rows = cursor.execute(sql, params)
16 conn.commit()
17 cursor.close()
18 conn.close()
19 print('affected_rows=%d' % affected_rows)
```

第 1 行：引入 pymysql 库。

第 3~10 行：建立数据库连接 conn。

第 12 行：从 conn 中创建数据库操作所需的游标 cursor。

第 13 行：定义数据删除 SQL 语句，整数也用%s 作为参数占位符。

第 14 行：定义参数的数值。

第 15 行：执行数据删除 SQL 语句。

第 16 行：将删除操作写入数据库中。

第 17 行：关闭游标 `cursor`，完成数据库操作后必须关闭游标。

第 18 行：关闭数据库连接，如果是数据库连接池，应该是将数据库连接放回数据库连接池。

第 19 行：打印删除的行数。

14.5.5 数据库查询操作

在数据库查询操作中，有可能查询结果有很多条记录，但是我们只需要其中的一部分，所以我们将讨论分页技术。在这个例子中，我们将查询用户表中 `user_id<1000` 的所有记录，代码如下：

```
1 import pymysql
2
3 conn = pymysql.connect(
4     host='localhost',
5     port=3306,
6     user='dlb',
7     passwd='dlb123',
8     db='DlbDb',
9     charset='utf8'
10 )
11
12 cursor = conn.cursor()
13 sql = 'select count(user_id) from t_user where user_id<%s'
14 params = (10000)
15 cursor.execute(sql, params)
16 rec = cursor.fetchone()
17 total = rec[0]
18
19 sql = 'select user_id, user_name, salary, create_date from t_user \
20     where user_id<%s limit 0,3'
21 params = (10000)
22 cursor.execute(sql, params)
23 rowcount = cursor.rowcount
24 for row in cursor.fetchall():
25     salary = 0.0
26     if row[2]:
27         salary = row[2]
28     print('%d, %s, %.2f, %s' % (row[0], row[1], salary, row[3]))
29 print('total:%d, get:%d' % (total, rowcount))
30 cursor.close()
31 conn.close()
```

第 1 行：引入 `pymysql` 库。

第 3~10 行：创建数据库连接 `conn`。

第 12 行：从 `conn` 生成数据库操作所需游标 `cursor`。

第 13 行：定义数据查询 SQL 语句，本句主要用于查询共有多少条记录，因为下面将用 `limit` 来取指定范围的记录，所以需要求出符合查询条件的总记录数。

第 14 行：给出查询条件，查询条件应与实际查询时相同。

第 15 行：执行数据库查询操作。

第 16 行：取回一条记录。

第 17 行：取出符合查询条件的总记录数。

第 19 行：定义查询语句，利用 `limit` 来取指定范围内的记录，在本例中，指定从第 1 条记录开始取记录，一共取 3 条。

第 21 行：为 SQL 语句中的查询参数赋值。

第 22 行：执行查询语句。

第 23 行：求出实际取回几条记录，因为要取回 3 条记录，如果数据库中只有 2 条记录，就只能取回 2 条记录。

第 24 行：对取回的记录进行循环。

第 25~27 行：如果 salary 字段不为空，将值赋给 salary 变量，如果为空则 salary 变量的值为 0.0。

第 28 行：打印记录内容。

第 29 行：打印总共有多少条记录，以及本次取回多少条记录。

第 30 行：关闭游标 cursor。完成数据库操作后必须关闭游标。

第 31 行：关闭数据库连接。如果是数据库连接池，应该将数据库连接放回数据库连接池。

14.5.6 数据库事务操作

保证数据库的参照完整性，是使用数据库管理系统的一个重要原因。从我们建立的测试表来看，建立一个新账户时必须将该账户与用户进行绑定，如果用户绑定失败，则这个账户也不应该被创建。所以说这两个表的操作构成一个数据库事务，要么成功要么失败，不能有中间状态。这虽然可以通过编程来检查，但是既麻烦又容易出错，还有可能被遗忘。所以数据库管理系统提供了数据库事务，帮我们来完成这一功能。在本例中，我们将以创建用户账户为例，分别演示成功和失败的例子，代码如下：

```
1 import pymysql
2
3 conn = pymysql.connect(
4     host='localhost',
5     port=3306,
6     user='dlb',
7     passwd='dlb123',
8     db='DlbDb',
9     charset='utf8'
10 )
11
12 def add_acct(cursor):
13     sql = 'insert into t_acct(acct_name) values(%s)'
14     params = ('a001')
15     affected_rows = cursor.execute(sql, params)
16     acct_id = cursor.lastrowid
17     return acct_id
18
19 def add_user_acct(cursor, user_id, acct_id):
20     sql = 'insert into t_user_acct(user_id, acct_id) values(%s, %s)'
21     params = (user_id, acct_id)
22     affected_rows = cursor.execute(sql, params)
23     return True
24
25 def add_user_acct_error(cursor, user_id, acct_id):
26     return False
27
28
```

```

29 cursor = conn.cursor()
30 acct_id = add_acct(cursor)
31 print('acct_id=%d' % acct_id)
32 user_id = 2
33 if (add_user_acct(cursor, user_id, acct_id)):
34     print('add acct_id=%d user_id=%d' % (acct_id, user_id))
35     conn.commit()
36 cursor.close()
37
38 user_id = 5
39 cursor = conn.cursor()
40 acct_id = add_acct(cursor)
41 if (add_user_acct_error(cursor, user_id, acct_id)):
42     conn.commit()
43 else:
44     print('fail to add: acct_id=%d, user_id=%d' % (acct_id, user_id))
45     conn.rollback()
46 cursor.close()
47 conn.close()

```

运行结果如下：

```

acct_id=7
add acct_id=7 user_id=2
fail to add: acct_id=8, user_id=5
mysql> select * from t_acct;
+-----+-----+
| acct_id | acct_name |
+-----+-----+
|      1 | a001      |
|      3 | a001      |
|      5 | a001      |
|      7 | a001      |
+-----+-----+

4 rows in set (0.00 sec)

mysql> select * from t_user_acct;
+-----+-----+
| user_id | acct_id |
+-----+-----+
|      2 |      1 |
|      2 |      1 |
|      2 |      5 |
|      2 |      7 |
+-----+-----+

4 rows in set (0.00 sec)

```

从运行结果可以看出，第一次向 `t_acct` 表插入记录后，`acct_id=7`，此时向 `t_user_acct` 表插入记录成功，即事务成功，因此在 `t_user_acct` 表中加入了 2 和 7 的记录。当再次向 `t_acct` 表插入记录后，`acct_id=8`，但是此时向 `t_user_acct` 表插入记录失败，此时数据库事务回滚，`t_acct` 表和 `t_user_acct` 表中均没有相应的记录，实现了数据库事务的原子性，从而保证了数据的完整性。

下面来看具体的代码实现。

第 1 行：引入 `pymysql` 库。

第 3~10 行：创建数据库连接 `conn`。

第 12 行：定义 `add_acct` 函数，向 `t_acct` 表中添加记录，参数为数据库的游标 `cursor`。

第 13 行：向 `t_acct` 表中插入记录的 SQL 语句。

第 14 行：为 SQL 语句中的参数提供数值。

第 15 行：执行 SQL 语句。

第 16 行：求出新加入记录的主键。

第 17 行：返回该主键值。

第 19 行：定义 `add_user_acct` 函数，参数 `cursor` 为数据库游标，参数 `user_id` 和 `acct_id` 为 `t_user_acct` 表相应的字段值。

第 20 行：定义 `t_user_acct` 表 `insert` 语句。

第 21 行：定义 `insert` 语句中参数 `user_id` 和 `acct_id` 的数值。

第 22 行：执行 `SQL` 语句。

第 23 行：返回函数的执行结果为成功。

第 25 行：定义 `add_user_acct_error` 函数，参数 `cursor` 为数据库游标，参数 `user_id` 和 `acct_id` 为 `t_user_acct` 表相应的字段值。

第 26 行：该函数不做任何操作，只是返回失败结果，用于测试数据库事务回滚。

第 29 行：通过 `conn` 生成数据库操作所需游标 `cursor`。

第 30 行：调用 `add_acct` 函数，向 `t_acct` 表中添加记录，并记录 `acct_id` 的值。

第 32 行：定义 `user_id` 的值。

第 33 行：调用 `add_user_acct` 函数，并以 `cursor`、`user_id` 和 `acct_id` 为参数。

第 34 行：`add_user_acct` 函数执行成功，打印成功信息。

第 35 行：提交数据库事务，同时更改 `t_acct` 和 `t_user_acct` 表中的内容。

第 36 行：关闭游标。

第 38 行：给 `user_id` 赋新值。

第 39 行：重新从 `conn` 中获取数据库操作所需游标。

第 40 行：调用 `add_acct` 函数，向数据库中插入记录，并将主键值赋给 `acct_id`。

第 41 行：调用 `add_user_acct_error` 函数。

第 42 行：如果调用成功，则提交修改结果。

第 43~45 行：如果调用失败，则回滚数据库事务，并打印失败信息。

第 46 行：关闭数据库游标。

第 47 行：关闭数据库连接，如果是数据库连接池，应该将数据库连接放回数据库连接池。

14.5.7 数据库连接池

在上面的程序中，每次进行数据库操作均会重新建立数据库连接。这种处理方式在单机软件中是适用的，但是在服务器程序中就显得不太合适了。首先，建立连接和关闭连接是一个相对较慢的操作，如果服务器并发量较大，开销就比较大了；其次，如果每个用户甚至每个操作都建立数据库连接，那么在并发量非常大的情况下，数据库服务器的负担会非常重。因此，在服务器软件设计中，都会采用数据库连接池技术。

在本节中，我们将带领读者实现一个超简单的小型数据库连接池。

首先要将数据库连接池作为一个共享的全局变量，定义一个 `app_global` 模块，代码如下：

```

1 import time
2 import threading
3 import pymysql
4
5 g_db_pool_num = 5
6 g_db_pool = []
7
8 db_pool_lock = threading.Lock()
9
10 class Db_Pool_Cleaner(threading.Thread):
11     def __init__(self, duration):
12         threading.Thread.__init__(self)
13         self.duration = duration
14
15     def run(self):
16         curr_time = time.time()
17         print('db_pool_cleaner begin')
18         db_pool_lock.acquire()
19         for item in g_db_pool:
20             use_time = item['use_time']
21             if curr_time - use_time > self.duration:
22                 item['state'] = 0
23                 item['use_time'] = 0
24         db_pool_lock.release()
25         print('db_pool_cleaner end')
26         time.sleep(self.duration+3)
27         self.run()
28
29 def create_db_conn():
30     return pymysql.connect(
31         host = 'localhost',
32         port = 3306,
33         user = 'dlb',
34         passwd = 'dlb123',
35         db = 'DlbDb',
36         charset = 'utf8'
37     )
38
39 db_pool_cleaner = Db_Pool_Cleaner(5)
40 def init_db_pool():
41     state = 0
42     use_time = 0
43     for idx in range(g_db_pool_num):
44         conn = create_db_conn()
45         g_db_pool.append({'idx': idx, 'conn': conn, 'state': state, \
46             'use_time': use_time})
47     db_pool_cleaner.start()
48
49 def get_db_connection():
50     print('acquire lock')
51     db_pool_lock.acquire()
52     time.sleep(3)
53     for item in g_db_pool:
54         if (0 == item['state']):
55             item['state'] = 1
56             item['use_time'] = time.time()
57             print('release lock')
58             db_pool_lock.release()
59     return item
60     db_pool_lock.release()
61
62 def close_db_connection(conn_obj):
63     conn_obj['state'] = 0
64     conn_obj['use_time'] = 0

```

第 1 行：引入 `time` 库，需要记录每个连接被获取的时间，隔一段时间检查一遍连接池中的连接，如果获取时间超过一定的时限，就强制回收该连接，主要是防止程序获取连接池连接之后忘记关闭连接。

第 2 行：引入 `threading` 库，主要用于启动数据库连接池清理线程，处理数据库连接池并发访问锁机制。

第 3 行：引入 `pymysql` 用于数据库操作。

第 5 行：定义数据库连接池数量，由于是共享机制，所以不用设置过多，5~20 是比较合适的数量。

第 6 行：存放数据库连接的列表。

第 10 行：数据库连接池清理类，其是一个线程的子类。

第 11 行：定义构造函数，参数 `duration` 为连接池中连接允许的最大存在时间，超过该时间将强制回收连接。

第 12 行：调用父类的构造函数。

第 13 行：定义 `duration` 属性来存放连接允许使用最大时间。

第 15 行：线程启动函数。

第 16 行：记录当前时间，距 1970 年的秒数。

第 18 行：获取数据库连接列表对象的锁，防止多线程同时访问时出现问题。

第 19 行：对数据库连接列表中的连接循环执行第 20~23 行操作。

第 20 行：针对每个数据库连接列表中的连接，求出连接获取时间。

第 21 行：如果连接使用时间大于允许时间，则执行第 22、23 行操作。

第 22 行：将连接状态置为空闲。

第 23 行：将连接获取时间置为 0。

第 24 行：循环完成后，释放数据库连接列表对象锁。

第 26 行：暂停指定时间。

第 27 行：重新执行数据库连接池清理工作。

第 29~37 行：定义生成数据库连接函数，返回新建立的数据库连接。

第 39 行：建立数据库连接池清理对象。

第 40 行：定义初始化数据库连接池函数。

第 41 行：初始时数据库连接池中连接的状态为空闲。

第 42 行：初始时数据库连接池中连接的获取时间为 0。

第 43 行：循环建立 `g_db_pool_num` 个连接，并放入连接池中。

第 44 行：调用 `create_db_conn` 创建新的数据库连接。

第 45 行：将新建立的连接生成一个字典对象，保存到数据库连接列表中。

第 47 行：启动数据库连接池清理线程。

第 49 行：定义获取数据库连接池中的连接函数。

第 51 行：先获取数据库连接池锁。

第 52 行：这是为了方便演示加入的，调用 `Sleep` 函数暂停 3 秒，在实际应用中不应该有这段代码。

第 53 行：循环数据库连接池中的连接。

第 54 行：如果该数据库连接使用状态为空闲，执行第 55~59 行操作。

第 55 行：将该连接状态置为已用。

第 56 行：记录该连接获取时间。

第 58 行：释放数据库连接池锁。

第 59 行：返回该数据库连接字典对象。

第 62 行：定义关闭数据库连接函数，参数为数据库连接字典对象（由 `get_db_connection` 函数获取到的）。

第 63 行：将该连接使用状态置为空闲。

第 64 行：将该连接获取时间置为 0。

有了数据库连接池的简单实现之后，再来看在主程序中怎样使用数据库连接池，代码如下：

```
1 import time
2 import pymysql
3 import app_global as ag
4
5 import threading
6 class db_pool_thread(threading.Thread):
7     def __init__(self, thread_id, name):
8         threading.Thread.__init__(self)
9         self.thread_id = thread_id
10        self.name = name
11
12    def run(self):
13        print('start %s' % self.name)
14        conn_obj = ag.get_db_connection()
15        print('%s: %s' % (self.name, conn_obj))
16        #ag.close_db_connection(conn_obj)
17
18 ag.init_db_pool()
19
20
21 thd1 = db_pool_thread(101, 'c1')
22 thd2 = db_pool_thread(102, 'c2')
23
24 thd1.start()
25 thd2.start()
26 conn_obj = ag.get_db_connection()
27 print('main: %s' % conn_obj)
28 thd1.join()
29 thd2.join()
30
31 print('bye')
32
33 for conn_obj in ag.g_db_pool:
34     print(conn_obj)
35
36 ag.db_pool_cleaner.join()
```

第 6 行：定义 `db_pool_thread` 类，用于模拟在多线程状态下获取数据库连接池的情况。

第 7 行：定义构造函数，参数为线程 ID 和名称，这里主要用于调试时的后台输出。

第 8 行：调用父类的构造函数。

第 9 行：定义属性 `thread_id` 并赋值。

第 10 行：定义属性 `name` 并赋值。

第 12 行：定义线程启动函数。

第 13 行：打印自己的名字，主要用于调试。

第 14 行：调用 `get_db_connection` 类获取数据库连接对象。

第 15 行：打印获取到的数据库连接对象。

第 18 行：初始化数据库连接池。

第 21 行：创建第一个数据库连接池获取线程对象。

第 22 行：创建第二个数据库连接池获取线程对象。

第 24 行：启动第一个数据库连接池获取线程。

第 25 行：启动第二个数据库连接池获取线程。

第 26 行：调用 `get_db_connection` 类获取数据库连接池连接对象。

第 27 行：打印获取到的数据库连接池连接对象。

第 28 行：等待第一个数据库连接池线程结束。

第 29 行：等待第二个数据库连接池线程结束。

第 33、34 行：打印当前数据库连接池状态。

第 36 行：等待数据库连接池清理线程结束。

程序运行结果如下：

```
1 db_pool_cleaner begin
2 db_pool_cleaner end
3 start c1
4 acquire lock
5 start c2
6 acquire lock
7 acquire lock
8 release lock
9 c1: {'state': 1, 'use_time': 1483581649.6875114, 'conn': <pymysql.connection
s.Connection object at 0x7ff124430438>, 'idx': 0}
10 release lock
11 main: {'state': 1, 'use_time': 1483581652.6921957, 'conn': <pymysql.connecti
ons.Connection object at 0x7ff12270b978>, 'idx': 1}
12 db_pool_cleaner begin
13 release lock
14 c2: {'state': 1, 'use_time': 1483581655.6957273, 'conn': <pymysql.connection
s.Connection object at 0x7ff12270bc88>, 'idx': 2}
15 db_pool_cleaner end
16 bye
17 {'state': 1, 'use_time': 1483581649.6875114, 'conn': <pymysql.connections.Co
nnection object at 0x7ff124430438>, 'idx': 0}
18 {'state': 1, 'use_time': 1483581652.6921957, 'conn': <pymysql.connections.Co
nnection object at 0x7ff12270b978>, 'idx': 1}
19 {'state': 1, 'use_time': 1483581655.6957273, 'conn': <pymysql.connections.Co
nnection object at 0x7ff12270bc88>, 'idx': 2}
20 {'state': 0, 'use_time': 0, 'conn': <pymysql.connections.Connection object a
t 0x7ff12270bda0>, 'idx': 3}
21 {'state': 0, 'use_time': 0, 'conn': <pymysql.connections.Connection object a
t 0x7ff12270beb8>, 'idx': 4}
22 db_pool_cleaner begin
23 db_pool_cleaner end
24 db_pool_cleaner begin
25 db_pool_cleaner end
```

第 1、2 行：在一开始启动程序，初始化数据库连接池时，启动数据库连接池清理线程打印的日志。

第 3、4 行：启动第一个数据库连接获取线程打印的日志，由于其在获取数据库连接池锁后会暂停 3 秒，因此线程会停下来。

第 5、6 行：启动第二个数据库连接获取线程打印的日志，由于其无法获得数据库连接池的锁，因此线程会停下来。

第 7 行：主线程获取数据库连接打印的日志。

第 8 行：第一个数据库连接获取线程释放数据库连接池锁。

第 9 行：打印获取到的数据库连接。

第 10 行：主线程获取到数据库连接池锁暂停 3 秒后，释放数据库连接池锁。

第 11 行：主线程打印获取到的数据库连接。

第 12 行：数据库连接池清理线程请求数据库连接池锁，由于此时数据库连接池锁已经

被第二个数据库连接获取线程得到，因此必须等待。

第 13 行：第二个数据库连接获取线程释放数据库连接池锁。

第 14 行：第二个数据库连接获取线程打印获取到的数据库连接对象。

第 15 行：数据库连接池清理线程获取到数据库连接池锁，清理数据库连接池中的超时连接对象。

第 17~21 行：打印数据库连接池当前状态。

第 22~25 行：每隔固定时间，数据库连接池清理线程会清除数据库连接池中的超时连接对象。

需要注意的是，上面所打印的日志只是在一台机器上运行一次打印的内容。读者如果运行这个程序，可能会打印出不同的结果，但是原理是一样的，读者也可以做出类似的分析。

14.6 任务队列

深度学习服务与普通的 Web 应用还是有一些区别的，因为是计算密集型业务，有一些内容可能在短时间内无法完成；或者即使可以完成，也会因为资源限制变得不现实。因此需要同步和异步两种调用机制，对于耗时短的服务，可以采用同步方式调用，立刻返回结果给客户端；而对于耗时较长的服务，可以采用异步调用方式，将用户的请求放入队列中统一进行处理，请求完成后，再通过邮件、短信等途径通知用户查看结果。同步方式与一般的 Web 应用没有区别，这里就不再讨论了。但是异步方式与普通的 Web 应用有较大的区别，在这一节里将重点讨论这一问题。

异步任务体系架构如图 14.5 所示。

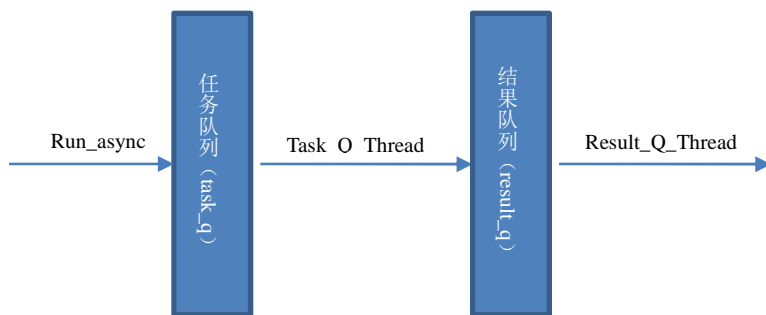


图 14.5 异步任务体系架构

如图 14.5 所示，当调用异步任务时，首先将任务加入任务队列 `task_q` 中。任务队列线程 `Task_Q_Thread` 为一个独立的线程，循环从任务队列中取出任务并运行该任务，得到任务结果后将结果加入结果队列中。结果队列线程 `Result_Q_Thread` 从结果队列中取出结果，通过合适的方式通知用户任务运行结果。

之所以将任务放入一个队列，由一个线程来串行执行，主要是为了提高运行效率。如

果按照通常的做法，每个任务建立一个线程，由于每个任务耗时较长，系统就会有很多线程存在，需要在线程间频繁切换。而线程间上下文转换，对我们来说是无用功，因此性能会降低很多。解决这种问题最好的方法就是采用任务队列方式，采用单线程来运行。可能有些读者对这种方式有疑问，认为多线程才是性能最好的。我们可以举一个实例，如 Node.js 就是采用这种策略，因此性能上傲视绝大部分服务器，比很多用 C++写的服务器性能都要好，因为其他服务器采用的是线程池，而 Node.js 采用异步任务。

异步任务的部分代码如下：

```
1 import threading
2 import queue
3
4 global task_q
5 global result_q
6
7 class Task_Q_Thread(threading.Thread):
8     def __init__(self):
9         threading.Thread.__init__(self)
10
11     def run(self):
12         params = task_q.get(block=True)
13         while params:
14             print('get task from task_q')
15             result = {'status': 'Ok', 'out': [1, 1, 1], \
16                     'user_id': params['user_id']}
17             result_q.put(result)
18             print('result enqueue')
19             params = task_q.get(block=True)
20
21 class Result_Q_Thread(threading.Thread):
22     def __init__(self):
23         threading.Thread.__init__(self)
24
25     def run(self):
26         result = result_q.get(block=True)
27         while result:
28             print('process result:%s' % result)
29             result = result_q.get(block=True)
30
31 def init_dl_manager():
32     global task_q, result_q
33     task_q = queue.Queue(maxsize=10)
34     result_q = queue.Queue(maxsize=10)
35     task_q_thread = Task_Q_Thread()
36     task_q_thread.start()
37     result_q_thread = Result_Q_Thread()
38     result_q_thread.start()
39
40 def run_sync(params):
41     print('run_sync: %s' % params)
42     result = {'status': 'Ok', 'out': [1, 0, 1], 'user_id': params['user_id']}
43     return result
44
45 def run_async(params):
46     global task_q
47     print('run_async: %s' % params)
48     task_q.put(params)
49     return True
```

第 1 行：引入多线程库 `threading`。

第 2 行：引入队列库。

第 4 行：定义全局变量 `task_q`，用于存放异步任务。

第 5 行：定义全局变量 `result_q`，用于存入异步任务完成后的结果。

第 7 行：定义 `Task_Q_Thread` 类，异步任务执行线程类。

第 8、9 行：定义构造函数。

第 11 行：定义线程启动函数。

第 12 行：从异步任务队列中取出最早的任务。

第 13 行：如果任务不为空，则执行第 14~19 行操作。

第 15 行：通过执行这个异步任务业务逻辑，产生结果字典对象。

第 17 行：将结果加入结果队列中。

第 19 行：取下一条异步任务队列中的任务，重新回到第 13 行判断循环是否应该结束。

第 21 行：定义 `Result_Q_Thread` 类，结果队列异步执行类。

第 22、23 行：定义构造函数。

第 25 行：定义线程启动函数。

第 26 行：从结果队列中取出最早的结果。

第 27 行：如果结果不为空，则循环执行第 28、29 行操作。

第 28 行：打印结果内容，实际应用中应该将结果通知用户，可以通过邮件、短信、微信等方式通知。

第 29 行：从结果队列列表中读取下一条结果。

第 40~43 行：定义同步任务，这是调用深度学习服务时的默认方式。此时直接运行该任务，并返回执行结果给调用者。

第 45 行：定义 `run_async` 函数，以异步的方式执行任务，只针对特别耗时的深度学习服务，例如 K 最小近邻法等。

第 46 行：获取全局变量 `task_q`。

第 48 行：将任务加入异步任务队列中。

第 49 行：返回成功，向调用者表明已经接收到任务，请等待运行结果。

完成异步任务体系架构定义之后，再来看怎样使用这个异步系统。注意，我们在这一章中只是介绍具体实现技术，下一章将把这些技术组合在一起，开发一个真正的深度学习云平台。代码如下：

```
1 import time
2 import dl_manager as dl
3
4 dl.init_dl_manager()
5
6 ann_task = {'algorithm': 'logistic_regression', 'x': 1, 'y': 1.1, 'user_id':
7             1001}
8 result = dl.run_sync(ann_task)
9 print('sync result: %s' % result)
10
11 if dl.run_async(ann_task):
12     print('call run successfully please wait for result')
13 else:
14     print('fail to run')
```

第 2 行：引入异步任务系统。

第 4 行：初始化异步任务系统。

第 6 行：生成一个运行深度学习算法的任务，这里是运行逻辑回归算法，包括算法所需输入信号、哪个用户请求的这个任务等信息。

第 8、9 行：以同步方式运行这个任务，即时返回运行结果，并打印运行结果。

第 11~14 行：以异步任务方式运行这个任务，将其加入异步任务队列，等待系统运行这个任务。

运行这段程序，应该得到如下所示的运行结果：

```
1 run_sync: {'algorithm': 'logistic_regression', 'x': 1, 'y': 1.1, 'user_id': 1001}
2 sync result: {'status': 'Ok', 'user_id': 1001, 'out': [1, 0, 1]}
3 run_async: {'algorithm': 'logistic_regression', 'x': 1, 'y': 1.1, 'user_id': 1001}
4 call run successfully please wait for result
5 get task from task_q
6 result enqueue
7 process result: {'status': 'Ok', 'user_id': 1001, 'out': [1, 1, 1]}
```

第 1、2 行：如果是同步任务，可以立即得到运行结果。

第 3 行：如果以异步任务方式运行，将被加入异步任务队列中。

第 4 行：程序立即返回异步任务接收是否成功的结果。注意，此时任务还在异步任务队列中排队，很可能还没有开始执行。

第 5 行：异步任务队列线程从异步任务队列中取出任务，并执行该任务。

第 6 行：将得到的结果加入结果队列中。

第 7 行：结果队列线程从结果队列中取出结果，通过各种方式通知调用者任务运行的结果。

14.7 媒体文件上传

在深度学习中，有一些样本的数量特别大，如图像，如果以数值方式传递，一方面数据量很大，另一方面还要求调用者按照特定的格式预处理数据，这提高了服务的使用门槛。因此，对于图像识别、声音识别等领域，如果允许用户直接上传媒体文件，我们在后台进行预处理，并生成所需输入信号格式，这将是一个比较合理的解决方案。

下面就来看一下在 CherryPy 框架下，怎样实现媒体文件的上传和下载，代码如下：

```
1 import sys
2 sys.path.append('./cherrypy')
3 import cherrypy
4 import cgi
5 import tempfile
6 import os
7 import shutil
8 import time
9
10 upload_dir = '/home/osboxes/dev/wky/work/book/chp14/upload/'
11
12 class FieldStorage(cgi.FieldStorage):
13     def make_file(self, binary=None):
14         return tempfile.NamedTemporaryFile()
15
16 def no_body_process():
17     cherrypy.request.process_request_body = False
18
19 cherrypy.tools.noBodyProcess = cherrypy.Tool('before_request_body', \
20                                             no_body_process)
```

```

21
22 class File_Uploader(object):
23     @cherry.py.expose
24     def index(self, params={}):
25         return '''
26 <html>
27 <body>
28 <form action="upload" method="POST" enctype="multipart/form-data">
29   File:<input type="file" name="theFile" /><br />
30   <input type="submit" />
31 </form>
32 </body>
33 </html>
34     '''
35     @cherry.py.expose
36     @cherry.py.tools.noBodyProcess()
37     def upload(self, theFile=None):
38         cherry.py.response.timeout = 3600
39         lcHDRS = {}
40         print('#### %s' % type(cherry.py.request.headers))
41         for hdr in cherry.py.request.headers.items():
42             lcHDRS[hdr[0].lower()] = hdr[1]
43         formFields = Field_Storage(fp=cherry.py.request.rfile, \
44                                   headers=lcHDRS, environ={'REQUEST_METHOD': \
45                                                           'POST'}, keep_blank_values=True)
46         theFile = formFields['theFile']
47         srcFile = theFile.file.name
48         print('##### %s' % theFile.filename)
49         destFile = upload_dir + '/f_' + \
50                   str(time.time()).replace('.', '_') + \
51                   os.path.splitext(theFile.filename)[1]
52         shutil.copy(srcFile, destFile)
53         return theFile.filename
54
55 cherry.py.server.socket_timeout = 60
56 cherry.py.server.max_request_body_size = 0
57
58 cherry.py.config.update({
59     'server.socket_host': '192.168.1.16',
60     'server.socket_port': 8090,
61 })
62 cherry.py.quickstart(File_Uploader(), '/', {'/': {}})

```

本例中只实现了文件上传代码的基本功能，所以 Web 服务器只是一个极简的例程，代码解析如下。

第 1~3 行：引入 CherryPy 库。

第 4~8 行：引入文件上传所需库。

第 10 行：定义文件上传的保存目录。

第 12 行：定义 Field_Storage 类，继承自 cgi.FieldStorage 类。

第 13、14 行：定义 Overload 父类的 make_file 方法，生成一个命名的文件。

第 16、17 行：配置 CherryPy 不处理消息体内容。

第 19 行：将定义的 no_body_process 函数挂接到 CherryPy 消息体处理之前。

第 22 行：定义文件上传类 File_Uploader。

第 23 行：定义其是 CherryPy 的可访问 URL。

第 24 行：定义 index 函数，显示默认页面。

第 25~34 行：定义一个极简的 Web 页面，其中有一个文件上传框和一个上传按钮。

第 35 行：定义其为 CherryPy 可访问的 URL，如图 14.6 所示。



图 14.6CherryPy 可访问的 URL

- 第 36 行：不进行消息体处理。
- 第 37 行：定义 `upload` 函数，处理上传文件操作。
- 第 38 行：设置最长上传时间。
- 第 39 行：定义字典 `lcHDRS`，保存 HTTP Headers 信息。
- 第 41、42 行：将 HTTP Headers 以键值对形式存放到 `lcHDRS` 中。
- 第 43~45 行：求出页面窗体中的所有域，保存到 `formFields` 中。
- 第 46 行：取出窗体中文件上传控件 `thefile` 的内容。
- 第 47 行：求出上传文件对应的临时文件的全路径文件名。
- 第 49~51 行：求出上传后的文件名，文件以 `f_` 开头，中间为 `time.time()` 的时间，将小数点替换为 “_”，扩展名为原文件的扩展名。
- 第 52 行：将上传文件从临时位置复制到上传目录中。
- 第 53 行：返回上传文件的原始文件名。

14.8 Redis 操作

在实际项目中，除了使用 MySQL 数据库，为了提高性能，还会经常使用 NoSQL 数据库，其中用得最多的是 Redis。Redis 不仅具有键值对形式的缓存，还具有断电恢复的功能，也提供了比键值对丰富的数据操作，在实际项目中得到了广泛应用。

我们在这一节里将简要地向读者介绍一下 Redis 的用法，先介绍 Redis 服务器的安装和配置，再介绍一个简单的例程，向读者演示 Redis 的使用方法。

14.8.1 Redis 安装配置

Redis 的安装配置方法有很多种，最简单的要属下载源码编译安装了。因为 Redis 不需要任何依赖库的配置，而且连 `configure` 的过程都可以省略，编译过程中不会出现需要第三方库的情况，这在开源软件中绝对是非常少见的。

先到网站上下载最新源码版本，运行以下命令进行编译：

```
wget https://github.com/antirez/redis/archive/4.0-rc2.tar.gz
tar -xzf 4.0-rc2.tar.gz
cd redis-4.0-rc2
make
```

编译完成后，再进行安全性配置，编辑源码目录下的 `redis.conf` 文件，修改以下两处：

```
66 # IF YOU ARE SURE YOU WANT YOUR INSTANCE TO LISTEN TO ALL THE INTERFACES
67 # JUST COMMENT THE FOLLOWING LINE.
68 #
69 bind 192.168.1.16

84 # By default protected mode is enabled. You should disable it only if
85 # you are sure you want clients from other hosts to connect to Redis
86 # even if no authentication is configured, nor a specific set of interfaces
87 # are explicitly listed using the "bind" directive.
88 protected-mode no
```

在上面的代码中，第一处是对第 69 行代码的修改，是让 Redis 只在 192.168.1.16 这个 IP 地址上监听；第二处是对第 88 行代码的修改，是让 Redis 关闭安全模式，否则只有本机才能访问 Redis。

修改完成之后，利用以下命令启动 Redis：

```
(wky) osboxes@osboxes:~/dev/redis-4.0-rc2$ src/redis-server redis.conf &
```

14.8.2 Redis 使用例程

在使用 Redis 功能之前需要安装 Redis 库：

```
(wky) osboxes@osboxes:~/dev/redis-4.0-rc2$ pip install redis
```

下面来看 Redis 操作的具体程序，代码如下：

```
1 import redis
2
3 redis_pool = redis.ConnectionPool(host='192.168.1.16', port=6379)
4 redis_obj = redis.Redis(connection_pool=redis_pool)
5 r1 = redis.Redis(connection_pool=redis_pool)
6 redis_obj.set('user_name', 'wky001')
7
8 user_name = redis_obj.get('user_name')
9 print('user_name=%s' % user_name)
10 u_n = r1.get('user_name')
11 print('u_n=%s' % u_n)
```

第 1 行：引入 Redis 库。

第 3 行：定义 Redis 连接池，只需提供 IP 地址和端口即可。

第 4 行：从 Redis 连接池获取一个 Redis 对象 `redis_obj`。

第 5 行：从 Redis 连接池再获取一个 Redis 对象 `r1`。

第 6 行：通过 `redis_obj` 设置 `user_name` 键的值为 `wky001`。

第 8、9 行：获取 `user_name` 键的值，并打印该值。

第 10、11 行：从 `r1` 上获取 `user_name` 键的值，并打印该值。

由上面的程序可以看出，Redis 的使用比数据库要简单得多。而且 Redis 的性能优于数据库系统，所以在实际中应用更多。

至此，我们介绍了深度学习云平台建设所需的全部 Web 开发技术。在本章中，我们比较注重基本技术的使用，并没有特别结合应用场景，也没有过多考虑系统架构问题。在下一章中，我们会利用本章介绍的 Web 开发技术，建立一个完整的深度学习云平台，并且以利用多层感知器算法判断 MNIST 手写数字识别问题为例，讲解为此算法建立一个深度学习云平台的具体步骤。

第 15 章

深度学习云平台

在上一章中，我们对基于 Python 3.x 深度学习服务云平台 Web 实现技术进行了详细讲解。在这一章中，我们将利用上一章中讲到的技术，搭建一个小型完备的深度学习服务云平台系统。我们先以 MNIST 手写数据集数字识别为例，讲解怎样将机器学习算法改造为深度学习服务形式；接着讲述怎样将这个深度学习云服务连接到 Web 服务器上；最后再进行开放性讨论，进入生产阶段的深度学习算法，讲解怎样进行调优和持续改进。

15.1 神经网络持久化

想要让深度学习算法成为深度学习服务，首先需要做的就是将深度学习网络的参数和超参数进行持久化存储，这样才能保证不必每次重启计算机时都重新学习。我们知道，深度学习网络训练是十分耗时的操作，在通常情况下，重新学习一遍训练数据都不具有现实可行性。即使要学习，最好也在以前的基础上进行学习。所以神经网络参数和超参数持久化就显得非常重要了。

15.1.1 数据库表设计

为了更好地表示深度学习网络状态，可以采用如图 15.1 所示的实体关系模型。

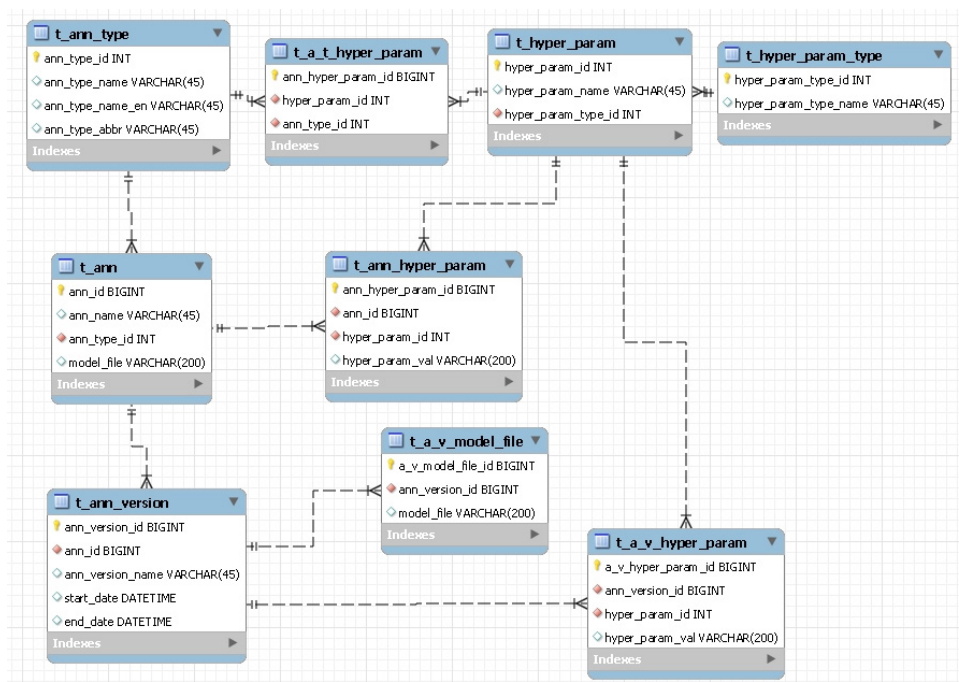


图 15.1 深度学习网络实体关系模型

下面对各数据库表结构进行说明。

1. 神经网络类型表（t_ann_type）

保存当前所有类型的神经网络。

字段名	名称	类型	主/外键	备注
ann_type_id	ann类型编号	int	PK	
ann_type_name	中文名称	varchar		
ann_type_name_en	英文名称	varchar		
ann_type_abbr	英文简称	varchar		

初始化数据为：

```
1 INSERT INTO `dlbDb`.`t_ann_type` (`ann_type_id`, `ann_type_name`,
2   `ann_type_name_en`, `ann_type_abbr`) VALUES (1, '线性回归',
3   'linear regression', 'LNRG');
4 INSERT INTO `dlbDb`.`t_ann_type` (`ann_type_id`, `ann_type_name`,
5   `ann_type_name_en`, `ann_type_abbr`) VALUES (2, '逻辑回归',
6   'logistic regression', 'LGRG');
7 INSERT INTO `dlbDb`.`t_ann_type` (`ann_type_id`, `ann_type_name`,
8   `ann_type_name_en`, `ann_type_abbr`) VALUES (3, '支撑向量机',
9   'support vector machine', 'SVM');
10 INSERT INTO `dlbDb`.`t_ann_type` (`ann_type_id`, `ann_type_name`,
11   `ann_type_name_en`, `ann_type_abbr`) VALUES (4, '多层感知器',
12   'multi-layer percpetron', 'MLP');
13 INSERT INTO `dlbDb`.`t_ann_type` (`ann_type_id`, `ann_type_name`,
14   `ann_type_name_en`, `ann_type_abbr`) VALUES (5, '卷积神经网络',
15   'convolutional neural network', 'CNN');
16 INSERT INTO `dlbDb`.`t_ann_type` (`ann_type_id`, `ann_type_name`,
17   `ann_type_name_en`, `ann_type_abbr`) VALUES (6, '递归神经网络',
18   'recurrent neural network', 'RNN');
19 INSERT INTO `dlbDb`.`t_ann_type` (`ann_type_id`, `ann_type_name`,
20   `ann_type_name_en`, `ann_type_abbr`) VALUES (7, '长短时记忆',
```



```
21 'long short term memory', 'LSTM');
22 INSERT INTO `Dlbbdb`.`t_ann_type` (`ann_type_id`, `ann_type_name`,
23 `ann_type_name_en`, `ann_type_abbr`) VALUES (8, '去噪自动编码器',
24 'denoising auto encoder', 'dA');
25 INSERT INTO `Dlbbdb`.`t_ann_type` (`ann_type_id`, `ann_type_name`,
26 `ann_type_name_en`, `ann_type_abbr`) VALUES (9, '稀疏自动编码器',
27 'sparse auto encoder', 'SA');
28 INSERT INTO `Dlbbdb`.`t_ann_type` (`ann_type_id`, `ann_type_name`,
29 `ann_type_name_en`, `ann_type_abbr`) VALUES (10, '堆叠去噪自动编码器',
30 'stack denosing auto encoder', 'SdA');
31 INSERT INTO `Dlbbdb`.`t_ann_type` (`ann_type_id`, `ann_type_name`,
32 `ann_type_name_en`, `ann_type_abbr`) VALUES (11, '堆叠稀疏自动编码器',
33 'stack sparse auto encoder', 'SSA');
34 INSERT INTO `Dlbbdb`.`t_ann_type` (`ann_type_id`, `ann_type_name`,
35 `ann_type_name_en`, `ann_type_abbr`) VALUES (12, '受限波尔兹曼机',
36 'restrict bolzman machine', 'RBM');
37 INSERT INTO `Dlbbdb`.`t_ann_type` (`ann_type_id`, `ann_type_name`,
38 `ann_type_name_en`, `ann_type_abbr`) VALUES (13, '深度信念网络',
39 'deep believe network', 'DBN');
40 INSERT INTO `Dlbbdb`.`t_ann_type` (`ann_type_id`, `ann_type_name`,
41 `ann_type_name_en`, `ann_type_abbr`) VALUES (14, '高斯判决分析',
42 'gaussian discrimetive analysis', 'GDA');
43 INSERT INTO `Dlbbdb`.`t_ann_type` (`ann_type_id`, `ann_type_name`,
44 `ann_type_name_en`, `ann_type_abbr`) VALUES (15, '朴素贝叶斯分析',
45 'noive bayes analysis', 'NB');
46 INSERT INTO `Dlbbdb`.`t_ann_type` (`ann_type_id`, `ann_type_name`,
47 `ann_type_name_en`, `ann_type_abbr`) VALUES (16, '变分自动编码器',
48 'variation auto encoder', 'VAE');
49 INSERT INTO `Dlbbdb`.`t_ann_type` (`ann_type_id`, `ann_type_name`,
50 `ann_type_name_en`, `ann_type_abbr`) VALUES (17, '生成对抗网络',
51 'generative adversary network', 'GAN');
52 INSERT INTO `Dlbbdb`.`t_ann_type` (`ann_type_id`, `ann_type_name`,
53 `ann_type_name_en`, `ann_type_abbr`) VALUES (18, '神经图灵机',
54 'neural turning machine', 'NTM');
```

2. 超参数类型表

目前，维护超参数的类型共有三种情况：整数、实数、字符串。

字段名	名称	类型	主/外键	备注
hyper_param_type_id	超参数类型编号	int	PK	
hyper_param_type_name	超参数类型名称			

初始化数据为：

```
1 INSERT INTO `Dlbbdb`.`t_hyper_param_type` (`hyper_param_type_id`,
2 `hyper_param_type_name`) VALUES (1, '整数');
3 INSERT INTO `Dlbbdb`.`t_hyper_param_type` (`hyper_param_type_id`,
4 `hyper_param_type_name`) VALUES (2, '实数');
5 INSERT INTO `Dlbbdb`.`t_hyper_param_type` (`hyper_param_type_id`,
6 `hyper_param_type_name`) VALUES (3, '字符串');
```

3. 超参数表（t_hyper_param）

保存所有神经网络用到的超参数名称和类型。

字段名	名称	类型	主/外键	备注
hyper_param_id	超参数编号	int	PK	
hyper_param_name	超参数名称	varchar		
hyper_param_type_id	超参数类型编号	int	FK	T_hyper_param_type表外键

初始化数据为：

```
1 INSERT INTO `Dlbb`.`t_hyper_param` (`hyper_param_id`, `hyper_param_name`,
2   `hyper_param_type_id`) VALUES (1, '学习率', 2);
3 INSERT INTO `Dlbb`.`t_hyper_param` (`hyper_param_id`, `hyper_param_name`,
4   `hyper_param_type_id`) VALUES (2, '层数', 1);
5 INSERT INTO `Dlbb`.`t_hyper_param` (`hyper_param_id`, `hyper_param_name`,
6   `hyper_param_type_id`) VALUES (3, '每层神经元数', 3);
7 INSERT INTO `Dlbb`.`t_hyper_param` (`hyper_param_id`, `hyper_param_name`,
8   `hyper_param_type_id`) VALUES (4, '动量项', 2);
9 INSERT INTO `Dlbb`.`t_hyper_param` (`hyper_param_id`, `hyper_param_name`,
10  `hyper_param_type_id`) VALUES (5, 'L1', 2);
11 INSERT INTO `Dlbb`.`t_hyper_param` (`hyper_param_id`, `hyper_param_name`,
12  `hyper_param_type_id`) VALUES (6, 'L2', 2);
```

4. 神经网络类型超参数表 (t_a_t_hyper_param)

保存每个神经网络类型所需超参数信息，这个表是一个模板，用于生成每个具体神经网络超参数表实例。

字 段 名	名 称	类 型	主/外键	备 注
ann_hyper_param_id	主键	bigint	PK	
hyper_param_id	超参数编号	int	FK	T_hyper_param表外键
ann_type_id	ann类型编号	int	FK	T_ann_type表外键

初始化参数：不同神经网络类型所需超参数不同，本例只以多层感知器模型为例。

```
1 INSERT INTO `Dlbb`.`t_a_t_hyper_param` (`ann_hyper_param_id`,
2   `hyper_param_id`, `ann_type_id`) VALUES (1, 1, 4);
3 INSERT INTO `Dlbb`.`t_a_t_hyper_param` (`ann_hyper_param_id`,
4   `hyper_param_id`, `ann_type_id`) VALUES (2, 2, 4);
5 INSERT INTO `Dlbb`.`t_a_t_hyper_param` (`ann_hyper_param_id`,
6   `hyper_param_id`, `ann_type_id`) VALUES (3, 3, 4);
7 INSERT INTO `Dlbb`.`t_a_t_hyper_param` (`ann_hyper_param_id`,
8   `hyper_param_id`, `ann_type_id`) VALUES (4, 4, 4);
9 INSERT INTO `Dlbb`.`t_a_t_hyper_param` (`ann_hyper_param_id`,
10  `hyper_param_id`, `ann_type_id`) VALUES (5, 5, 4);
11 INSERT INTO `Dlbb`.`t_a_t_hyper_param` (`ann_hyper_param_id`,
12  `hyper_param_id`, `ann_type_id`) VALUES (6, 6, 4);
```

5. 神经网络实例表 (t_ann)

表示具体运行的神经网络。

字 段 名	名 称	类 型	主/外键	备 注
ann_id	神经网络编号	bigint	PK	
ann_name	神经网络名称	varchar		
ann_type_id	神经网络类型编号	int	FK	T_ann_type表外键
model_file	模型文件	varchar		保存模型权值参数和超参数

初始化数据为：

```
1 INSERT INTO `Dlbb`.`t_ann` (`ann_id`, `ann_name`, `ann_type_id`,
2   `model_file`) VALUES (1, 'MNIST手写数字识别MLP', 4, 'best_model.pkl');
```

6. 神经网络超参数表 (t_ann_hyper_param)

保存具体运行的神经网络超参数的具体值。

字 段 名	名 称	类 型	主/外键	备 注
ann_hyper_param_id	主键	bigint	PK	

续表

字 段 名	名 称	类 型	主/外键	备 注
ann_id	神经网络编号	bigint	FK	T_ann表外键
hyper_param_id	超参数编号	int	FK	T_hyper_param表外键
hyper_param_val	超参数值	varchar		

初始化数据为:

```
1 INSERT INTO `DlBdb`.`t_ann_hyper_param` (`ann_hyper_param_id`, `ann_id`,
2 `hyper_param_id`, `hyper_param_val`) VALUES (1, 1, 1, '0.01');
3 INSERT INTO `DlBdb`.`t_ann_hyper_param` (`ann_hyper_param_id`, `ann_id`,
4 `hyper_param_id`, `hyper_param_val`) VALUES (2, 1, 2, '3');
5 INSERT INTO `DlBdb`.`t_ann_hyper_param` (`ann_hyper_param_id`, `ann_id`,
6 `hyper_param_id`, `hyper_param_val`) VALUES (3, 1, 3, '[784, 500, 10]');
7 INSERT INTO `DlBdb`.`t_ann_hyper_param` (`ann_hyper_param_id`, `ann_id`,
8 `hyper_param_id`, `hyper_param_val`) VALUES (4, 1, 4, '0');
9 INSERT INTO `DlBdb`.`t_ann_hyper_param` (`ann_hyper_param_id`, `ann_id`,
10 `hyper_param_id`, `hyper_param_val`) VALUES (5, 1, 5, '0');
11 INSERT INTO `DlBdb`.`t_ann_hyper_param` (`ann_hyper_param_id`, `ann_id`,
12 `hyper_param_id`, `hyper_param_val`) VALUES (6, 1, 6, '0.0001');
```

7. 神经网络版本表 (t_ann_version)

深度学习网络需要持续改进，因此会产生很多逐渐改进的版本。每个版本都会有自己的模型文件和超参数，本表用于维护这些信息。

字 段 名	名 称	类 型	主/外键	备 注
ann_version_id	ann版本编号	bigint	PK	
ann_id	ann编号	bigint	FK	T_ann表外键
ann_version_name	网络版本名称	varchar		
start_date	开始日期	datetime		默认值为创建时刻
end_date	结束日期	datetime		默认值为2999-12-31

初始化数据为:

```
1 INSERT INTO `DlBdb`.`t_ann_version` (`ann_version_id`, `ann_id`,
2 `ann_version_name`, `start_date`, `end_date`) VALUES (1, 1, 'v1.0',
3 '2017-01-06', '2999-12-31');
```

8. 神经网络版本模型文件表 (t_a_v_model_file)

保存神经网络每个版本对应的模型文件。

字 段 名	名 称	类 型	主/外键	备 注
a_v_model_file_id	主键	bigint	PK	
ann_version_id	ann版本编号	bigint	FK	T_ann_version表外键
model_file	模型文件名	varchar		模型文件名的命名规则为: bm_为前缀, 加上ann_version_id, 后缀为.pkl

初始化数据为:

```
1 INSERT INTO `DlBdb`.`t_a_v_model_file` (`a_v_model_file_id`,
2 `ann_version_id`, `model_file`) VALUES (1, 1, 'bm_1.pkl');
```

9. 神经网络版本超参数表 (t_a_v_hyper_param)

字 段 名	名 称	类 型	主/外键	备 注
a_v_hyper_param_id	主键	bigint	PK	
ann_version_id	ann版本编号	bigint	FK	T_ann_version表外键
hyper_param_id	超参数编号	int	FK	T_hyper_param表外键
hyper_param_val	超参数值	varchar		

初始化数据为:

```
1 INSERT INTO `dlb`.`t_a_v_hyper_param` (`a_v_hyper_param_id`,
2   `ann_version_id`, `hyper_param_id`, `hyper_param_val`)
3   VALUES (1, 1, 1, '0.01');
4 INSERT INTO `dlb`.`t_a_v_hyper_param` (`a_v_hyper_param_id`,
5   `ann_version_id`, `hyper_param_id`, `hyper_param_val`)
6   VALUES (2, 1, 2, '3');
7 INSERT INTO `dlb`.`t_a_v_hyper_param` (`a_v_hyper_param_id`,
8   `ann_version_id`, `hyper_param_id`, `hyper_param_val`)
9   VALUES (3, 1, 3, '[784, 500, 10]');
10 INSERT INTO `dlb`.`t_a_v_hyper_param` (`a_v_hyper_param_id`,
11   `ann_version_id`, `hyper_param_id`, `hyper_param_val`)
12   VALUES (4, 1, 4, '0');
13 INSERT INTO `dlb`.`t_a_v_hyper_param` (`a_v_hyper_param_id`,
14   `ann_version_id`, `hyper_param_id`, `hyper_param_val`)
15   VALUES (5, 1, 5, '0');
16 INSERT INTO `dlb`.`t_a_v_hyper_param` (`a_v_hyper_param_id`,
17   `ann_version_id`, `hyper_param_id`, `hyper_param_val`)
18   VALUES (6, 1, 6, '0.0001');
```

15.1.2 整体目录结构

由于要开发一个深度学习服务云平台，所以需要仔细规划目录结构，如图 15.2 所示。



图 15.2 程序整体目录结构

下面对各个文件目录的功能做一下简单描述。

- ann: 保存所有深度学习网络模型的源码。
- ann/mlp: 保存多层感知器模型的源码。
- conf: 系统配置信息，如数据库等。
- controller: Web 服务器的控制器层。
- data: 模型的训练数据和用户上传的数据。

lib: 第三方库源码目录。

lib/cherrypy: CherryPy 框架源码。

lib/theano: Theano 框架源码。

logs: 记录系统日志。

model: MVC 中的模型类，主要用于操作数据库或 Redis。

public: 存放静态文件内容，如 JS、CSS、Images 等。

repository: 保存神经网络模型文件。

upload: 保存用户上传的文件。

view: 视图类，保存 HTML 文件。

work: 运行过程中临时用于存放信息的目录。

15.1.3 训练过程及模型文件保存

在这一节中，我们从应用启动开始分析整个多层感知器训练过程。先从主程序入口讲起，系统启动程序是从 `app_main` 开始的：

```
1 import sys
2 sys.path.append('./lib/Theano')
3 sys.path.append('./lib/cherrypy')
4 import os
5 import theano
6 import cherrypy
7 import app_global as ag
8 import model.m_mysql as db
9 import controller.c_mlp as mlp
10
11 if __name__ == '__main__':
12     print('starting up...')
13     db.init_db_pool()
14     ann_id = 1
15     mlp.train(ann_id)
16     db.rdb_pool_cleaner.join()
17     db.wdb_pool_cleaner.join()
```

第 1 行：引入 `sys` 库。

第 2、3 行：将 `Theano`、`CherryPy` 源码目录加入系统路径中，以便于包含。

第 4 行：引入 `os` 库。

第 5 行：引入 `theano` 库，用于深度学习。

第 6 行：引入 `cherrypy` 库，用于开发基于接口的 `Web` 服务器。

第 7 行：引入 `app_global` 模块，其中保存有全局变量。

第 8 行：引入 `m_mysql` 模块，该模块中有数据库连接池和数据库操作。

第 9 行：引入多层感知器模型控制器 `c_mlp`，这里采用 MVC 架构。

第 13 行：调用 `db.init_db_pool` 初始化数据库连接池。

第 14 行：指定神经网络 ID，通过这个 ID 可以找到神经网络所对应的超参数、模型文件、当前版本等信息。

第 15 行：调用多层感知器控制器的 `train` 方法，对网络进行训练。

第 16 行：等待读数据库连接池清理线程结束。

第 17 行：等待写数据库连接池清理线程结束。

这里引入了读/写分离技术，数据库读操作通过读操作数据库连接池来操作，数据库写操作通过写操作数据库连接池来操作。因此需要有两个数据库连接池清理线程。

全局变量存入模块 `app_conf` 的代码如下：

```
1 import os
2
3 dataset_dir = os.getcwd() + '/data/'
4 ann_mf_dir = os.getcwd() + '/repository/'
```

目前该模块中只定义了数据集存放目录 `dataset_dir` 和神经网络模型文件存放目录 `ann_mf_dir`。

下面来看比较复杂的数据库连接池和数据库操作模块 `m_mysql`。在这个模块里，我们采用了读/写分离技术，代码如下：

```
1 import time
2 import threading
3 import pymysql
4 import app_global as ag
5 import conf.app_conf as conf
6
7 rdb_pool_num = 5
8 rdb_pool = []
9 rdb_pool_lock = threading.Lock()
10
11 wdb_pool_num = 5
12 wdb_pool = []
13 wdb_pool_lock = threading.Lock()
14
15 class Db_Pool_Cleaner(threading.Thread):
16     def __init__(self, db_pool, db_pool_lock, duration, interval):
17         threading.Thread.__init__(self)
18         self.db_pool_lock = db_pool_lock
19         self.db_pool = db_pool
20         self.duration = duration
21         self.interval = interval
22
23     def run(self):
24         curr_time = time.time()
25         self.db_pool_lock.acquire()
26         for item in self.db_pool:
27             use_time = item['use_time']
28             if curr_time - use_time > self.duration:
29                 item['state'] = 0
30                 item['use_time'] = 0
31         self.db_pool_lock.release()
32         time.sleep(self.duration + self.interval)
33         self.run()
34
35 def create_rdb_conn():
36     return create_db_conn(conf.rdb)
37
38 def create_wdb_conn():
39     return create_db_conn(conf.wdb)
40
41 def create_db_conn(db):
42     return pymysql.connect(
43         host = db['host'],
44         port = db['port'],
45         user = db['user'],
46         passwd = db['passwd'],
47         db = db['db'],
48         charset = db['charset']
49     )
50
51 def _init_db_pool(db_pool, db_pool_num, create_db_conn_func):
52     state = 0
```

```

53     use_time = 0
54     for idx in range(db_pool_num):
55         conn = create_db_conn_func()
56         db_pool.append({'idx': idx, 'conn': conn, 'state': state, \
57                        'use_time': use_time})
58
59 rdb_pool_cleaner = Db_Pool_Cleaner(rdb_pool, rdb_pool_lock, 3600, 100)
60 wdb_pool_cleaner = Db_Pool_Cleaner(wdb_pool, wdb_pool_lock, 3600, 100)
61 def init_db_pool():
62     _init_db_pool(rdb_pool, rdb_pool_num, create_rdb_conn)
63     _init_db_pool(wdb_pool, wdb_pool_num, create_wdb_conn)
64     rdb_pool_cleaner.start()
65     wdb_pool_cleaner.start()
66
67 def _get_db_connection(db_pool, db_pool_lock):
68     db_pool_lock.acquire()
69     for item in db_pool:
70         if (0 == item['state']):
71             item['state'] = 1
72             item['use_time'] = time.time()
73             db_pool_lock.release()
74             return item
75     db_pool_lock.release()
76     return null
77
78 def close_db_connection(conn_obj):
79     conn_obj['state'] = 0
80     conn_obj['use_time'] = 0
81
82 def get_rdb_connection():
83     return _get_db_connection(rdb_pool, rdb_pool_lock)
84
85 def get_wdb_connection():
86     return _get_db_connection(wdb_pool, wdb_pool_lock)
87
88 def query(sql, params):
89     conn_obj = get_rdb_connection()
90     conn = conn_obj['conn']
91     result = query_t(conn, sql, params)
92     close_db_connection(conn_obj)
93     return result
94
95 def query_t(conn, sql, params):
96     cursor = conn.cursor()
97     cursor.execute(sql, params)
98     rowcount = cursor.rowcount
99     rows = cursor.fetchall()
100    cursor.close()
101    return (rowcount, rows)

```

第 1 行：引入 time 库，记录时间点和时间间隔。

第 2 行：引入 threading 库，处理多线程和并发问题。

第 3 行：引入 pymysql 库，处理数据库相关操作。

第 4 行：引入 app_global 模块，使用其中的全局变量。

第 5 行：引入 app_conf 模块，主要使用其中的读、写数据库连接配置信息。

第 7~9 行：定义读操作数据库连接池相关变量，rdb_pool_num 为连接池数量，rdb_pool 为数据库连接对象列表，rdb_pool_lock 为读数据库连接池锁。为了防止在多线程情况下对数据库连接池操作引发冲突，对数据库连接池操作均先要获取数据库连接池锁，这样其他线程再想获取数据库连接池锁就必须等待本线程操作结束。这样做虽然使性能有些损失，但是可以避免多线程编程中容易造成的数据操作冲突问题。

第 11~13 行：定义写操作数据库连接池的相关变量，wdb_pool_num 为写连接池的数量，wdb_pool 为数据库连接对象列表，wdb_pool_lock 为写数据库连接池锁。

第 15 行：定义 Db_Pool_Cleaner 类，是数据库连接池清理线程类，负责定期对数据库

连接池进行清理，清除获取时间超限后仍然没有关闭的数据库连接池中的连接，其是 `threading.Thread` 类的子类。

第 16 行：定义构造函数，`db_pool` 为数据库连接列表，可以是读连接池也可以是写连接池；`db_pool_lock` 为数据库连接池锁，同样可以是读连接池锁也可以是写连接池锁；`duration` 表示允许连接最大打开时间，超过该时间将关闭；`interval` 表示在最大打开时间过后多长时间重新检查。

第 17 行：初始化父类构造函数。

第 18 行：定义属性 `db_pool_lock` 保存数据库连接池锁。

第 19 行：定义属性 `db_pool` 保存数据库连接池。

第 20 行：定义属性 `duration` 保存连接最大允许打开时间。

第 21 行：定义属性 `interval` 最大打开时间多久后重新执行检查。

第 23 行：定义线程启动函数。

第 24 行：记录当前时间 `curr_time`。

第 25 行：获取连接池锁，可以是读数据库连接池也可以是写数据库连接池。

第 26 行：对数据库连接池中的连接对象循环第 27~30 行操作。

第 27 行：取出数据库连接对象的获取时间。

第 28 行：如果获取时间大于允许的最大打开时间，则执行第 29 和 30 行操作。

第 29 行：将状态置为空闲。

第 30 行：将获取时间设为 0。

第 31 行：清理完数据库连接池超时连接对象后，释放数据库连接池锁。

第 32 行：休眠时打开最大允许时间和时间间隔。

第 33 行：休眠过后再次执行本函数，清理超时数据库连接对象。

第 35、36 行：创建只读数据库连接函数。

第 38、39 行：创建写入数据库连接函数。

第 41 行：定义数据库连接具体创建函数。

第 42~49 行：调用 `pymysql.connect` 函数，并以 `app_conf` 里面的数据库连接参数为参数。

第 51 行：定义数据库连接池初始化内部函数，`db_pool` 为数据库连接对象列表，`db_pool_num` 为数据库连接池数量，`create_db_conn_func` 为数据库连接创建函数指针。

第 52 行：数据库连接对象默认状态为空闲。

第 53 行：数据库连接对象默认获取时间为 0。

第 54 行：循环生成 `db_pool_num` 个数据库连接，并将其放入数据库连接对象列表 `db_pool` 中。

第 55 行：根据数据库连接创建函数指针，调用相应的数据库连接创建函数，创建数据库连接。

第 56 行：生成字典变量表示的数据库连接对象，并加入数据库连接对象列表 `db_pool` 中。

第 59 行：创建读数据库连接池清理线程类对象。

第 60 行：创建写数据库连接池清理线程类对象。

第 61 行：定义初始化数据库连接池接口函数。注意，我们规定，函数如果以 “_” 开头，则代表内部函数，应该仅限于本模块内使用，非 “_” 开头的函数才可以作为接口供其他模块调用。

第 62 行：调用内部初始化数据库连接池函数，初始化读数据库连接池。

第 63 行：调用内部初始化数据库连接池函数，初始化写数据库连接池。

第 64 行：启动读数据库连接池清理线程。

第 65 行：启动写数据库连接池清理线程。

第 67 行：定义内部函数，获取数据库连接池中的空闲连接函数。

第 68 行：获取对应的数据库连接池锁，如果未能获取则等待。

第 69 行：成功获取数据库连接池锁之后，循环查看数据库连接池中的连接对象。

第 70 行：如果某个连接对象状态为空闲，则执行第 71~74 行操作。

第 71 行：将该数据库连接对象的状态设置为使用中。

第 72 行：将该数据库连接对象的获取时间设置为当前时间。

第 73 行：释放数据库连接池锁。

第 74 行：返回该数据库连接对象。

第 75 行：如果所有数据库连接池中的数据库连接对象的状态均不为空闲，释放数据库连接池锁。

第 76 行：返回空。

第 78 行：定义关闭数据库连接函数，参数为数据库连接对象。

第 79 行：将数据库连接对象状态设置为空闲。

第 80 行：将数据库连接对象获取时间设置为 0。

第 82、83 行：获取只读数据库连接函数，以读数据库连接对象列表和读数据库连接池锁为参数，调用内部获取数据库连接函数。

第 85、86 行：获取写数据库连接函数，以写数据库连接对象列表和写数据库连接池锁为参数，调用内部获取数据库连接函数。

第 88 行：定义数据库查询函数，sql 为带参数的查询 SQL 语句，params 为含有参数值的元组。

第 89 行：获取读数据库连接对象。

第 90 行：取出其中的数据库连接。

第 91 行：调用 query_t 执行 SQL 语句，参数再加上一个数据库连接 conn。

第 92 行：关闭数据库连接。

第 93 行：返回数据库查询结果。

第 95 行：定义数据库查询函数，conn 为数据库连接，sql 为带参数的查询 SQL 语句，params 为参数值元组。之所以定义这个函数，是因为如果数据库操作是在一个事务中进行的，我们希望在同一个数据库连接上执行 SQL 操作，因此在事务中时直接调用本函数，而不在事务中时直接调用 query 函数。

第 96 行：在 conn 上创建游标。

第 97 行：以 SQL 语句和参数元组为参数，在游标上执行 `execute` 函数。

第 98 行：求出本次取到的记录数。

第 99 行：取出所有结果集赋给 `rows`。

第 100 行：关闭游标。

第 101 行：返回本次取出记录数和查询结果集元组。

下面来看神经网络超参数模型类，模型类主要用于从数据库中读取内容，该类负责取出神经网络的超参数数值，代码如下：

```
1 import model.m_mysql as db
2
3 def get_ann_hyper_params(ann_id):
4     sql = 'select hyper_param_id, hyper_param_val from t_ann_hyper_param \
5           where ann_id=%s'
6     params = (ann_id)
7     rowcount, rows = db.query(sql, params)
8     if rowcount != 8:
9         print('error:')
10        return {}
11    ahp = {}
12    for row in rows:
13        ahp[str(row[0])] = row[1]
14    return ahp
```

第 3 行：定义获取神经网络超参数的函数，`ann_id` 为神经网络编号。

第 4、5 行：定义带参数占位符的 SQL 查询语句。

第 6 行：定义 SQL 中参数值的元组。

第 7 行：调用 `m_mysql.query` 函数，求出查询结果，包括本次取到的行数和结果集。

第 8~10 行：如果取得记录数小于 8，则说明超参数不全，返回空值。

第 11 行：定义超参数字典。

第 12、13 行：以 `hyper_param_id` 为键，以 `hyper_param_val` 为值创建超参数字典。

第 14 行：返回超参数字典。

下面来看多层感知器的控制器类，代码如下：

```
1 import model.m_ann_hyper_param as m_ahp
2 import model.m_ann_version as m_av
3 from ann.mlp.mlp_mnist_engine import MlpMnistEngine
4
5 def get_ahps(ann_id):
6     raw_ahps = m_ahp.get_ann_hyper_params(ann_id)
7     ahps = {}
8     ahps['learning_rate'] = float(raw_ahps['1'])
9     ahps['layers'] = int(raw_ahps['2'])
10    ahps['layer_nums'] = raw_ahps['3']
11    ahps['momentum'] = float(raw_ahps['4'])
12    ahps['L1'] = float(raw_ahps['5'])
13    ahps['L2'] = float(raw_ahps['6'])
14    ahps['batch_size'] = int(raw_ahps['7'])
15    ahps['n_epochs'] = int(raw_ahps['8'])
16    return ahps
17
18 def get_ann_version(ann_id):
19     return m_av.get_ann_version(ann_id)
20
21 def train(ann_id):
22     ahps = get_ahps(ann_id)
23     ann_version_info = get_ann_version(ann_id)
24     ahps['ann_version_id'] = ann_version_info['ann_version_id']
25     ahps['dataset'] = 'mnist.pkl.gz'
26     mlp_engine = MlpMnistEngine(ahps)
27     mlp_engine.train()
```

第 1 行：引入神经网络超参数模型类，用于读取指定神经网络超参数值。

第 2 行：引入神经网络版本模型类，用于读取神经网络的最新版本。

第 3 行：引入多层感知器引擎类。

第 5 行：定义获取超参数函数，`ann_id` 为神经网络编号。

第 6 行：调用神经网络超参数模型类 `get_ann_hyper_params` 方法，获取该神经网络超参数字典。

第 8 行：获取学习率参数。

第 9 行：获取层数。

第 10 行：获取各层神经元数。

第 11 行：获取学习方法中的动量项参数。

第 12 行：获取 L1 调整项系数，L1 调整项指连接权值绝对值之和。

第 13 行：获取 L2 调整项系数，L2 调整项指连接权值的平方和。

第 14 行：获取迷你批次中的样本数量。

第 15 行：获取训练样本集遍历最大次数。

第 16 行：返回超参数字典。

第 18 行：定义获取神经网络最新版本的信息函数，`ann_id` 为神经网络编号。

第 19 行：调用神经网络版本模型类的获取神经网络最新版本信息函数，获取指定神经网络最新版本信息。

第 21 行：定义多层感知器训练方法。

第 22 行：获取多层感知器超参数。

第 26 行：创建多层感知器引擎类实例，以多层感知器超参数为参数。

第 27 行：调用多层感知器引擎类 `train` 方法，进行网络训练。

多层感知器引擎类我们在 4.5 节已经进行过解析，但是这里对原来的程序进行了重构，为了内容的完整性，再向读者重新讲解一遍，重构代码的目录结构如图 15.3 所示。

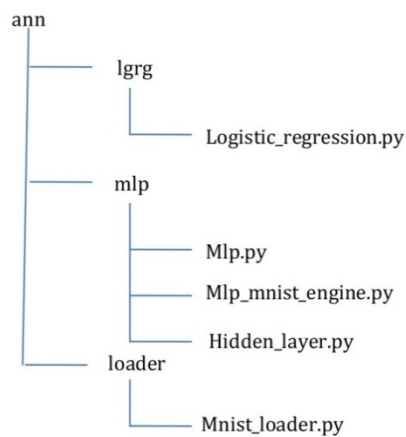


图 15.3 深度学习重构代码的目录结构

我们首先来看逻辑回归目录 `lgrg` 下的逻辑回归类 `LogisticRegression`，代码如下：

```

1 from __future__ import print_function
2 __docformat__ = 'restructuredtext en'
3
4 import six.moves.cPickle as pickle
5 import gzip
6 import os
7 import sys
8 import timeit
9 import numpy
10 import theano
11 import theano.tensor as T
12
13 class LogisticRegression(object):
14     def __init__(self, input, n_in, n_out):
15         self.W = theano.shared(
16             value=numpy.zeros(
17                 (n_in, n_out),
18                 dtype=theano.config.floatX
19             ),
20             name='W',
21             borrow=True
22         )
23         self.b = theano.shared(
24             value=numpy.zeros(
25                 (n_out,),
26                 dtype=theano.config.floatX
27             ),
28             name='b',
29             borrow=True
30         )
31         self.p_y_given_x = T.nnet.softmax(T.dot(input, self.W) + self.b)
32         self.y_pred = T.argmax(self.p_y_given_x, axis=1)
33         self.params = [self.W, self.b]
34         self.input = input
35
36     def negative_log_likelihood(self, y):
37         return -T.mean(T.log(self.p_y_given_x)[T.arange(y.shape[0]), y])
38
39     def errors(self, y):
40         if y.ndim != self.y_pred.ndim:
41             raise TypeError(
42                 'y should have the same shape as self.y_pred',
43                 ('y', y.type, 'y_pred', self.y_pred.type)
44             )
45         if y.dtype.startswith('int'):
46             return T.mean(T.neq(self.y_pred, y))
47         else:
48             raise NotImplementedError()

```

第 14 行：定义构造函数，其参数为输入信号、特征数量 n_{in} 、输出信号类别数量。

第 15~22 行：定义 Theano 共享变量，用于保存逻辑回归参数，即 θ_i ， $i \in \{1, 2, \dots, n\}$ ，并且初始化为 0。

第 23~30 行：定义 Theano 共享变量，用于保存逻辑回归参数 θ_0 。

第 31 行：定义 Theano 输出为 softmax 回归形式，因为输出有 10 个类别。

第 32 行：定义最终的类型是输出中值最大的一个。

第 33 行：定义我们的逻辑回归模型参数为 θ_0 和 θ_i ，其中 θ_i 为向量。

第 34 行：定义逻辑回归模型的输入为参数中指定的输入。

第 36、37 行：定义逻辑回归所用的代价函数为负对数似然函数，根据前文所介绍的理论知识，我们学习算法的目的就是使其最小化。在理论部分，讲述了我们使用似然函数作为逻辑回归算法的代价函数，为了计算方便，我们又将代价函数定义为对数似然函数，并求出了对数似然函数的最大值，从而从数学上推导出了逻辑回归学习算法。在这里，为了与神经网络学习算法统一，代价函数取为负对数似然函数（NLL），同时从求对数似然函数的最大值变为求负对数似然函数的最小值。

第 39~48 行：定义了误差计算函数。我们用该函数计算模型在样本集上的误差，用于衡量模型的性能。其定义为模型输出与正确输出的误差的平均值。

下面来看多层感知器类的定义，代码如下：

```
1 from __future__ import print_function
2 __docformat__ = 'restructuredtext en'
3
4 import os
5 import sys
6 import timeit
7 import numpy
8 import theano
9 import theano.tensor as T
10 from logistic_regression import LogisticRegression
11 from mnist_loader import MnistLoader
12 from hidden_layer import HiddenLayer
13
14 class MLP(object):
15     def __init__(self, rng, input, n_in, n_hidden, n_out):
16         self.hiddenLayer = HiddenLayer(
17             rng=rng,
18             input=input,
19             n_in=n_in,
20             n_out=n_hidden,
21             activation=T.tanh
22         )
23         self.logRegressionLayer = LogisticRegression(
24             input=self.hiddenLayer.output,
25             n_in=n_hidden,
26             n_out=n_out
27         )
28         self.L1 = (
29             abs(self.hiddenLayer.W).sum()
30             + abs(self.logRegressionLayer.W).sum()
31         )
32         self.L2_sqr = (
33             (self.hiddenLayer.W ** 2).sum()
34             + (self.logRegressionLayer.W ** 2).sum()
35         )
36         self.negative_log_likelihood = (
37             self.logRegressionLayer.negative_log_likelihood
38         )
39         self.errors = self.logRegressionLayer.errors
40         self.params = self.hiddenLayer.params +
41             self.logRegressionLayer.params
42         self.input = input
```

第 14 行：定义 MLP 类。

第 15 行：定义 MLP 类构造函数。

- ☐ rng: 随机数生成引擎。
- ☐ input: 输入向量。
- ☐ n_in: 输入向量维度，也是输入层神经元数目。
- ☐ n_hidden: 隐藏层神经元数目。
- ☐ n_out: 输出层神经元数目，也是模式分类问题中的类别数目。

第 16~22 行：定义隐藏层，其输入为网络的输入，输入向量维度为 `n_in`，隐藏层神经元数目为 `n_hidden`，激活函数为双曲正切函数。

第 23~27 行：定义输出层，其输入为隐藏层的输出，输入向量维度为隐藏层神经元数目，本层神经元数目为 `n_out`，即模式分类的类别数。

第 28~31 行：定义调整量 L1，其定义为所有连接权值绝对值之和。

第 32~35 行：定义调整量 L2，其定义为所有连接权值平方之和。

L1 和 L2 是代价函数的附加项，其目的是保证在优化过程中，优先取连接权值小的解决方案。

第 36~38 行：定义代价函数为逻辑回归模型的负对数似然函数。

第 39 行：定义逻辑回归层的误差为多层感知器模型的误差。

下面来看隐藏层类定义，代码如下：

```
1 from __future__ import print_function
2 __docformat__ = 'restructuredtext en'
3 import os
4 import sys
5 import timeit
6 import numpy
7 import theano
8 import theano.tensor as T
9 from logistic_regression import LogisticRegression
10
11 class HiddenLayer(object):
12     def __init__(self, rng, input, n_in, n_out, W=None, b=None,
13                 activation=T.tanh):
14         self.input = input
15         if W is None:
16             W_values = numpy.asarray(
17                 rng.uniform(
18                     low=-numpy.sqrt(6. / (n_in + n_out)),
19                     high=numpy.sqrt(6. / (n_in + n_out)),
20                     size=(n_in, n_out)
21                 ),
22                 dtype=theano.config.floatX
23             )
24             if activation == theano.tensor.nnet.sigmoid:
25                 W_values *= 4
26
27             W = theano.shared(value=W_values, name='W', borrow=True)
28
29         if b is None:
30             b_values = numpy.zeros((n_out,), dtype=theano.config.floatX)
31             b = theano.shared(value=b_values, name='b', borrow=True)
32
33         self.W = W
34         self.b = b
35
36         lin_output = T.dot(input, self.W) + self.b
37         self.output = (
38             lin_output if activation is None
39             else activation(lin_output)
40         )
41         # parameters of the model
42         self.params = [self.W, self.b]
```

第 11 行：定义隐藏层类 HiddenLayer。

第 12、13 行：定义隐藏层类 HiddenLayer 的构造函数。

- ☐ rng: 随机数生成引擎。
- ☐ input: 输入信号向量。
- ☐ n_in: 输入向量维度。
- ☐ n_out: 本层神经元数量。
- ☐ W: 本隐藏层到下一层的连接权值矩阵。
- ☐ b: 偏置值向量，其维度为 n_out。
- ☐ activation: 神经元的激活函数，默认值为双曲正切函数。在本例中采用双曲正切函数，在实际应用中也可以采用改进线性单元（ReLU）或泛化改进线性单元。

第 14 行：给输入向量属性赋值。

第 15~27 行代码处理连接权值矩阵。

第 15 行：如果构造函数中指定了连接权值矩阵，则直接采用；如果没有指定，则运行下面的程序，生成随机的初始权值矩阵。

第 16~23 行：利用 Theano 的数组生成权值矩阵，矩阵初始值采用均匀分布的随机数。

第 24、25 行：如果神经元激活函数为 Sigmoid 函数，则需要增大权值向量的值。前面我们讨论过，Sigmoid 函数可以与双曲正切函数相互转换，这两种函数从某种意义上来说是等价的，但是双曲正切函数在性能上优于 Sigmoid 函数，因此如果没有特别的原因，应该选择双曲正切函数。

第 27 行：将刚生成的权值矩阵定义为 Theano 的共享变量。

第 29~31 行：如果没有指定偏置值的值，将偏置值向量所有分量赋为 0，并将新生成的偏置值向量定义为 Theano 的共享变量。

第 36 行：定义 $W \cdot x + b$ 值为神经元的输入值。

第 37~40 行：定义本层输出值向量为 $f_{\text{activation}}(W \cdot x + b)$ 。

下面来看多层感知器引擎类，代码如下：

```
1 from __future__ import print_function
2 __docformat__ = 'restructuredtext en'
3 import six.moves.cPickle as pickle
4 import os
5 import sys
6 import timeit
7 import json
8 import numpy
9 import theano
10 import theano.tensor as T
11 from ann.lrg.logistic_regression import LogisticRegression
12 from ann.loader.mnist_loader import MnistLoader
13 from ann.mlp.mlp import MLP
14 import app_global as ag
15
16 class MlpMnistEngine(object):
17     def __init__(self, ahps):
18         print("create MlpMnistEngine")
19         self.learning_rate=ahps.get('learning_rate', 0.01)
20         self.L1_reg=ahps.get('L1', 0.00)
21         self.L2_reg=ahps.get('L2', 0.0001)
22         self.n_epochs=ahps.get('n_epochs', 1000)
23         self.batch_size=ahps.get('batch_size', 600) # 20
24         layer_nums = json.loads(ahps['layer_nums'])
25         self.n_in = int(layer_nums[0])
26         self.n_hidden=int(layer_nums[1])
27         self.n_out = int(layer_nums[2])
28         self.model_file = 'bf_%s.pkl' % ahps['ann_version_id']
29         self.dataset = ahps['dataset']
30
31     def build_model(self):
32         loader = MnistLoader()
33         datasets = loader.load_data(ag.dataset_dir + self.dataset)
34         train_set_x, train_set_y = datasets[0]
35         valid_set_x, valid_set_y = datasets[1]
36         test_set_x, test_set_y = datasets[2]
37         n_train_batches = train_set_x.get_value(borrow=True).shape[0] \
38             // self.batch_size
39         n_valid_batches = valid_set_x.get_value(borrow=True).shape[0] \
40             // self.batch_size
41         n_test_batches = test_set_x.get_value(borrow=True).shape[0] \
42             // self.batch_size
43         print('... building the model')
44         index = T.lscalar() # index to a [mini]batch
45         x = T.matrix('x') # the data is presented as rasterized images
46         y = T.ivector('y') # the labels are presented as 1D vector of
47         rnq = numpy.random.RandomState(1234)
```

```

48     # 全新运行时
49     classifier = MLP(
50         rng=rng,
51         input=x,
52         n_in=self.n_in,
53         n_hidden=self.n_hidden,
54         n_out=self.n_out
55     )
56     cost = (
57         classifier.negative_log_likelihood(y)
58         + self.L1_reg * classifier.L1
59         + self.L2_reg * classifier.L2_sqr
60     )
61     test_model = theano.function(
62         inputs=[index],
63         outputs=classifier.errors(y),
64         givens={
65             x: test_set_x[index * self.batch_size:(index + 1) \
66                 * self.batch_size],
67             y: test_set_y[index * self.batch_size:(index + 1) \
68                 * self.batch_size]
69         }
70     )
71     validate_model = theano.function(
72         inputs=[index],
73         outputs=classifier.errors(y),
74         givens={
75             x: valid_set_x[index * self.batch_size:(index + 1) \
76                 * self.batch_size],
77             y: valid_set_y[index * self.batch_size:(index + 1) \
78                 * self.batch_size]
79         }
80     )
81     gparams = [T.grad(cost, param) for param in classifier.params]
82     updates = [
83         (param, param - self.learning_rate * gparam)
84         for param, gparam in zip(classifier.params, gparams)
85     ]
86     train_model = theano.function(
87         inputs=[index],
88         outputs=cost,
89         updates=updates,
90         givens={
91             x: train_set_x[index * self.batch_size: (index + 1) \
92                 * self.batch_size],
93             y: train_set_y[index * self.batch_size: (index + 1) \
94                 * self.batch_size]
95         }
96     )
97     return (classifier, n_train_batches, n_valid_batches,
98           n_test_batches, train_model, validate_model, test_model)
99
100 def train(self):
101     classifier, n_train_batches, n_valid_batches, n_test_batches, \
102     train_model, validate_model, test_model = self.build_mod
103
104     el()
105     print('... training')
106     patience = 15000 # 10000 # look as this many examples regardless
107     patience_increase = 2 # wait this much longer when a new best is
108     improvement_threshold = 0.995 # a relative improvement
109     # of this much is
110     validation_frequency = min(n_train_batches, patience // 2)
111     best_validation_loss = numpy.inf
112     best_iter = 0
113     test_score = 0.
114     start_time = timeit.default_timer()
115     epoch = 0
116     done_loopping = False
117     while (epoch < self.n_epochs) and (not done_loopping):
118         epoch = epoch + 1
119         for minibatch_index in range(n_train_batches):
120             minibatch_avg_cost = train_model(minibatch_index)
121             iter = (epoch - 1) * n_train_batches + minibatch_index
122             if (iter + 1) % validation_frequency == 0:
123                 validation_losses = [validate_model(i) for i
124                                     in range(n_valid_batches)]

```



```

123         this_validation_loss = numpy.mean(validation_losses)
124         print(
125             'epoch %i, minibatch %i/%i, validation error %f %%'
126             %
127             (
128                 epoch,
129                 minibatch_index + 1,
130                 n_train_batches,
131                 this_validation_loss * 100.
132             )
133         )
134         if this_validation_loss < best_validation_loss:
135             if (
136                 this_validation_loss < best_validation_loss *
137                 improvement_threshold
138             ):
139                 patience = max(patience,
140                               iter * patience_increase)
141                 best_validation_loss = this_validation_loss
142                 best_iter = iter
143                 test_losses = [test_model(i) for i
144                               in range(n_test_batches)]
145                 test_score = numpy.mean(test_losses)
146                 with open(ag.ann_mf_dir + 'best_model.pkl', 'wb') as
f:
147                     pickle.dump(classifier, f)
148                     print(('##epoch %i, minibatch %i/%i, test error of '
149                           'best model %f %%') %
150                           (epoch, minibatch_index + 1, n_train_batches,
151                             test_score * 100.))
152                 if patience <= iter:
153                     done_looping = True
154                     break
155             end_time = timeit.default_timer()
156             print(('Optimization complete. Best validation score of %f %% '
157                   'obtained at iteration %i, with test performance %f %%') %
158                   (best_validation_loss * 100., best_iter + 1,
159                     test_score * 100.))
160             print(('The code for file ' +
161                   os.path.split(__file__)[1] +
162                   ' ran for %.2fm' % ((end_time - start_time) / 60.)),
163                   file=sys.stderr)
164
165         def run(self):
166             print("run mlp v3")
167             classifier = pickle.load(open(self.model_file, 'rb'))
168             predict_model = theano.function(
169                 inputs=[classifier.input],
170                 outputs=classifier.logRegressionLayer.y_pred
171             )
172             dataset = self.dataset
173             loader = MnistLoader()
174             datasets = loader.load_data(dataset)
175             test_set_x, test_set_y = datasets[2]
176             test_set_x = test_set_x.get_value()
177             predicted_values = predict_model(test_set_x[:10])
178             print("Predicted values for the first 10 examples in test set:")
179             print(predicted_values)

```

第 11 行：引入在 `ann/lgrg/` 目录下的逻辑回归类 `LogisticRegression`。

第 12 行：引入在 `ann/loader/` 目录下的 MNIST 手写数字识别数据集载入类 `MnistLoader`。

第 13 行：引入在 `ann/mlp/` 目录下的多层感知器类 `MLP`。

第 16 行：定义多层感知器引擎类 `MlpMnistEngine`。

第 17 行：定义多层感知器引擎类 `MlpMnistEngine` 构造函数。

第 19 行：设置学习率为 0.01。

第 20 行：调整量 `L1` 为所有连接权值绝对值之和，`self.L1_reg` 为该值在代价函数中的权重。

第 21 行：调整量 $L2$ 为所有连接权值平方和，`self.L2_reg` 为该值在代价函数中的权重。

第 22 行：定义最大迭代次数，每循环完所有训练样本集一次，计为一次迭代。

第 23 行：定义迷你批次的大小，根据前文的讨论，迷你批次学习同时具有批次学习和在线学习的优点。

第 24~27 行：定义每层神经元数量。

第 28 行：定义保存网络参数的模型文件。

第 29 行：定义训练用数据集文件。

第 31 行：定义创建模型方法 `build_model`。

第 32、33 行：生成 `MnistLoader` 对象并调用 `load_data` 方法，载入 MNIST 手写数字识别数据集。

第 34 行：获取训练样本集输入信号和输出标签结果。

第 35 行：获取验证样本集输入信号和输出标签结果，验证样本集是从训练样本集中取出的一部分样本（通常为 10% 左右）。在模型训练过程中，每过一段时间，就计算一下验证样本集上的误差率，如果有提高则继续训练，如果没有提高则终止训练过程。注意：验证样本集不参加模型学习训练过程，只用于评价模型的泛化性能。

第 36 行：获取测试样本集输入信号和输出标签结果，当模型训练过程结束后，计算测试样本集上的误差，评价模型的性能。

第 37~42 行：将训练样本集、验证样本集、测试样本集，根据迷你批次大小，分别求出共有多少迷你批次数量。

第 44 行：定义迷你批次的索引。

第 45 行：定义 Theano 变量 x 表示样本输入信号。

第 46 行：定义 Theano 变量 y 表示样本输出标签结果。

第 47 行：初始化随机数生成引擎。

第 49~55 行：初始化多层感知器模型类，输入信号为 784 (28×28) 维，因为 MNIST 手写数字样本图片为 28×28 黑白图片。输出层神经元数为 10，因为要分类 0~9 个数字。

第 56~60 行：定义多层感知器模型的代价函数，定义其为负对数似然函数加上 $L1$ 调整项和 $L2$ 调整项。 $L1$ 调整项为对应权重乘以所有连接权值的绝对值之和，在本例中因为其对应的权重为 0，所以本项不起作用； $L2$ 调整项为对应 $L2$ 权重乘以所有连接权值平方和，这两项的作用是限制模型取小的连接权值，从而使尽量多的特征参与到决策中来。

第 61~70 行：定义测试模型，首先定义测试样本集迷你批次第 i 个批次为输入，输出为测试样本集上计算出的误差，因为我们使用通用符号定义的多层感知器模型，在具体计算过程中需要使用不同的测试样本，所以使用 Theano 的 `given` 关键字，将公式中的变量替换为当前测试样本集对应的数据。

第 71~80 行：定义验证模型，首先定义验证样本集迷你批次第 i 个批次为输入，输出为验证样本集上计算出的误差，因为我们使用通用符号定义的多层感知器模型，在具体计算过程中需要使用不同的验证样本，所以使用 Theano 的 `given` 关键字，将公式中的变量替换为当前验证样本对应的数据。

第 81 行：定义代价函数对所有参数求偏导数。

第 82~85 行：定义模型参数调整规则，模型参数更新公式为： $\Delta W_{ij}^{(l)(r)} = -\alpha \frac{\partial \mathcal{L}}{\partial W_{ij}^{(l)(r)}}$,

这里没有采用动量项。

第 86~96 行：定义训练模型，输入信号为训练样本集迷你批次的某批次，输出为模型的代价函数，更新规则为第 82~85 行定义的参数更新规则，通过 Theano 关键字 `given` 指定计算时具体使用的数据为训练样本某个迷你批次。

第 97、98 行：返回分类器、训练样本集迷你批次数、验证样本集迷你批次数、测试样本集迷你批次数、训练模型、验证模型、测试模型。

第 100 行：定义训练方法 `train`。

第 101、102 行：调用创建模型方法 `build_model`，生成分类器、训练样本集、验证样本集、测试样本集、训练模型、验证模型、测试模型。

第 104 行：定义 `patience` 变量，规定多少次循环验证样本集上的误差没有改善就停止运行训练过程。根据我们的运行经验，如果此值取 5000，则在我的虚拟机上大约运行不到 1 个小时，验证样本集的精度可达到 7.5% 左右；如果取值 15000，则运行 7~8 个小时，验证样本集上的误差可以达到 3.0% 左右。

第 105 行：如果发现新的最佳结果，延长允许没有改进的循环次数。

第 106 行：验证样本集上的误差改进达到 0.5% 以上，才认为有明显改进。

第 108 行：规定进行多少次循环后，在验证样本集上计算误差，检查是否有改进，并更新在验证样本集上取得的最佳结果。

第 109 行：在验证样本集上取得的最佳误差结果。

第 110 行：记录在验证样本集上取得的最佳误差结果的迭代数。

第 111 行：记录在测试样本集上的误差结果。

第 112 行：记录开始时间。

第 113 行：记录循环整个训练样本集次数。

第 114 行：循环中止控制条件。

第 115 行：当训练样本集循环总次数小于我们规定的最大循环次数，并且循环控制变量为假时，用训练样本集进行训练。

第 116 行：增加训练样本集总迭代次数。

第 117 行：对训练样本集每个迷你批次进行循环。

第 118 行：由训练模型计算出训练样本集在本迷你批次上的代价函数值。

第 119 行：计算运行迷你批次的总次数。

第 120 行：如果迷你批次运行总次数达到验证频率的规定，运行验证样本集上的验证过程。

第 121~123 行：由验证模型计算验证样本集上的总误差，并调用 `numpy` 方法求出验证样本集上的平均误差。

第 124~133 行：打印本次验证过程中的结果。

第 134~138 行：如果本次在验证样本集上求出的误差小于当前所取得的验证样本集上的最小误差，则进行相关处理。

第 139、140 行：更新多少次迷你批次循环，验证样本集上误差没有明显改进就停止训练的条件。

第 141 行：更新验证样本集上已经取得的最小误差值。

第 142 行：记录取得最佳验证样本集误差的迭代数。

第 143~145 行：通过测试模型求出测试样本集上的总误差，利用 `numpy` 求出测试样本集上的平均误差。

第 146、147 行：将目前的模型参数保存到最佳模型文件中。

第 148~151 行：打印本次验证过程的详细信息。

第 152~154 行：如果超过最大迷你批次循环次数后，验证样本集上的误差仍然没有明显改进，则将是否继续循环标志置为 `False`。

第 155 行：求出结束时间。

第 156~163 行：打印本次模型训练的汇总信息。

第 165 行：定义运行方法 `run`。

第 167 行：从神经网络版本所对应的模型文件中读入模型中的连接权值等参数。

第 168~171 行：定义预测模型，输入为模型的输入信号，输出为模型的预测值。

第 173、174 行：通过 `MNIST` 数据载入类，载入 `MNIST` 数据集。在这里，我们就将 `MNIST` 数据集中的数据交给训练好的模型进行预测。

第 175、176 行：取出测试样本集数据。

第 177 行：取出测试样本集前 10 条数据，为预测模型进行预测。

第 178、179 行：打印预测结果。

最后定义位于 `ann/loader/` 目录下的 `MNIST` 手写数字识别数据集载入类 (`MnistLoader`)，代码如下：

```
1 from __future__ import print_function
2 __docformat__ = 'restructuredtext en'
3
4 import six.moves.cPickle as pickle
5 import gzip
6 import os
7 import sys
8 import timeit
9 import numpy
10 import theano
11 import theano.tensor as T
12
13 class MnistLoader(object):
14     def load_data(self, dataset):
15         data_dir, data_file = os.path.split(dataset)
16         if data_dir == "" and not os.path.isfile(dataset):
17             new_path = os.path.join(
18                 os.path.split(__file__)[0],
19                 "..",
20                 "data",
21                 dataset
22             )
23             if os.path.isfile(new_path) or data_file == 'mnist.pkl.gz':
24                 dataset = new_path
25         if (not os.path.isfile(dataset)) and data_file == 'mnist.pkl.gz':
26             from six.moves import urllib
27             origin = (
28                 'http://www.iro.umontreal.ca/~lisa/deep/data/\
29                 mnist/mnist.pkl.gz'
30             )
31             print('Downloading data from %s' % origin)
```

```

32     urllib.request.urlretrieve(origin, dataset)
33     print('... loading data')
34     # Load the dataset
35     with gzip.open(dataset, 'rb') as f:
36         try:
37             train_set, valid_set, test_set = pickle.load(f,
38                                                         encoding='latin1')
39         except:
40             train_set, valid_set, test_set = pickle.load(f)
41
42     def shared_dataset(data_xy, borrow=True):
43         data_x, data_y = data_xy
44         shared_x = theano.shared(numpy.asarray(data_x,
45                                                 dtype=theano.config.floatX),
46                                 borrow=borrow)
47         shared_y = theano.shared(numpy.asarray(data_y,
48                                                 dtype=theano.config.floatX),
49                                 borrow=borrow)
50         return shared_x, T.cast(shared_y, 'int32')
51
52     test_set_x, test_set_y = shared_dataset(test_set)
53     valid_set_x, valid_set_y = shared_dataset(valid_set)
54     train_set_x, train_set_y = shared_dataset(train_set)
55
56     rval = [(train_set_x, train_set_y), (valid_set_x, valid_set_y),
57           (test_set_x, test_set_y)]
58     return rval

```

第 1~11 行：引入所需的 Python 包，其中 cPickle 就是 Python 中用于对象序列化的类，我们用这个类来处理经 Python 序列化过的 MNIST 手写数字识别数据集。

第 15 行：解析出数据集文件名的目录名和文件名。

第 16~32 行：检查是否存在指定的数据集文件，如果不存在，则从网上直接下载该文件。

第 35 行：以二进制只读方式打开数据集文件。

第 37 行：以 latin1 编码格式打开数据集文件，从中读出训练样本集、验证样本集、测试样本集。

第 40 行：如果第 37 行的操作失败，则用其他编码集打开数据集文件，从中读出训练样本集、验证样本集、测试样本集。

第 42~50 行：定义函数，生成对应样本集的共享变量，用于 Theano 定义的函数中，因为 Theano 预编译函数只能使用共享变量形式。

第 52 行：解析出测试样本集的输入信号和输出信号。

第 53 行：解析出验证样本集的输入信号和输出信号。

第 54 行：解析出训练样本集的输入信号和输出信号。

第 56~58 行：形成分类器引擎类需要的样本集形式，并返回给分类器引擎类。

15.2 神经网络运行模式

当深度学习网络训练完成之后，就可以将训练好的深度学习网络部署到生产环境中，使其处于运行状态。此时深度学习网络只进行前向计算，不进行误差反向传播，不对连接权值等参数进行调整。

首先需要改造一下 controller/c_mlp.py 文件，使其增加 run 方法，代码如下：

```

1 import model.m_ann_hyper_param as m_ahp
2 import model.m_ann_version as m_av
3 from ann.mlp.mlp_mnist_engine import MlpMnistEngine
4
5 def get_ahps(ann_id):
6     raw_ahps = m_ahp.get_ann_hyper_params(ann_id)
7     ahps = {}
8     ahps['learning_rate'] = float(raw_ahps['1'])
9     ahps['layers'] = int(raw_ahps['2'])
10    ahps['layer_nums'] = raw_ahps['3']
11    ahps['momentum'] = float(raw_ahps['4'])
12    ahps['L1'] = float(raw_ahps['5'])
13    ahps['L2'] = float(raw_ahps['6'])
14    ahps['batch_size'] = int(raw_ahps['7'])
15    ahps['n_epochs'] = int(raw_ahps['8'])
16    return ahps
17
18 def get_ann_version(ann_id):
19     return m_av.get_ann_version(ann_id)
20
21
22 def create_mlp_engine(ann_id):
23     ahps = get_ahps(ann_id)
24     ann_version_info = get_ann_version(ann_id)
25     ahps['ann_version_id'] = ann_version_info['ann_version_id']
26     ahps['dataset'] = 'mnist.pkl.gz'
27     return MlpMnistEngine(ahps)
28
29 def train(ann_id):
30     mlp_engine = create_mlp_engine(ann_id)
31     mlp_engine.train()
32
33 def run(ann_id, samples):
34     mlp_engine = create_mlp_engine(ann_id)
35     return mlp_engine.predict(samples)

```

在这个模块中，第 1~21 行代码在前面已经讲解过，变化主要从第 22 行代码开始。

第 22 行：定义新函数，生成多层感知器模型类对象。

第 23 行：调用 `get_ahps` 函数获取神经网络超参数。

第 24 行：调用 `get_ann_version` 方法获取神经网络当前版本号。

第 26 行：指定数据集文件名。

第 27 行：生成并返回 `MlpMnistEngine` 类实例。

第 29 行：定义训练方法，参数 `ann_id` 为神经网络编号。

第 30 行：调用 `create_mlp_engine` 方法生成 `MlpMnistEngine` 类对象。

第 31 行：调用多层感知器对象训练方法。

第 33 行：定义 `run` 方法，参数 `ann_id` 为神经网络编号，`samples` 为需要预测的样本。

第 34 行：调用 `create_mlp_engine` 方法生成多层感知器对象 `mlp_engine`。

第 35 行：调用 `mlp_engine` 对象的 `predict` 方法，参数 `samples` 为需要预测的样本集，返回值为预测结果。假设要预测 3 个样本，分别为手写数字 8、7、6，则返回结果为 [8, 7, 6]。

接着来看多层感知器引擎类，代码如下：

```

181 def predict(self, samples):
182     classifier = pickle.load(open(ag.ann_mf_dir +
183                                self.model_file, 'rb'))
184     predict_model = theano.function(
185         inputs = [classifier.input],
186         outputs = classifier.logRegressionLayer.y_pred
187     )
188     return predict_model(samples)

```

第 181 行：定义预测函数 `predict` 方法，参数 `samples` 为需要预测的样本集。

第 182 行：在神经网络当前版本所对应的模型文件创建分类器。

第 184 行：定义预测模型为 `theano` 函数，输入为分类器的输入，输出为分类器的逻辑回归层的输出层。

第 188 行：以需要预测的样本集为参数，运行 `theano` 函数 `predict_model`，并返回结果。最后来看怎样使用这个预测函数，代码如下：

```
1 import model.m_mysql as db
2 import controller.c_mlp as mlp
3 import app_global as ag
4
5 def init_app():
6     db.init_db_pool()
7     ann_id = 1
8     ann_version_info = mlp.get_ann_version(ann_id)
9     print(ann_version_info)
10
11 def get_sample():
12     from ann.loader.mnist_loader import MnistLoader
13     loader = MnistLoader()
14     dataset = ag.dataset_dir + 'mnist.pkl.gz'
15     datasets = loader.load_data(dataset)
16     test_set_x, test_set_y = datasets[2]
17     test_set_x = test_set_x.get_value()
18     return test_set_x[:1]
19
20
21 def startup():
22     init_app()
23     sample = get_sample()
24     ann_id = 1
25     result = mlp.run(ann_id, sample)
26     print('The result:%d' % result)
27     db.rdb_pool_cleaner.join()
28     db.wdb_pool_cleaner.join()
```

第 5 行：定义应用初始化函数 `init_app`，主要用于初始化数据库连接池。

第 11 行：获取需要预测的样本，这里是从测试样本集中取出一个样本。

第 12 行：引入 MNIST 手写数字识别数据集。

第 13 行：生成 MNIST 数据集载入类对象。

第 14 行：指定数据集文件。

第 15 行：读取数据集内容。

第 16 行：获取测试样本集输入信号集和输出标签集。

第 17 行：获取输入信号集内容。

第 18 行：取第一个样本并返回。

第 21 行：定义启动函数。

第 22 行：调用 `init_app` 函数，初始化数据库连接池。

第 23 行：从 MNIST 手写数字识别数据集的测试样本集中取出一个样本。

第 24 行：指定神经网络编号。

第 25 行：调用多层感知器控制器的运行方法，预测样本结果。

第 26 行：打印预测结果。

第 27、28 行：等待数据库连接池清理线程结束。

运行上述程序，会产生如下结果：

```
{'ann_version_name': 'v1.0', 'end_date': datetime.datetime(2999, 12, 31, 0, 0),
'start_date': datetime.datetime(2017, 1, 6, 0, 0), 'ann_version_id': 1}
... loading data
create MlpMnistEngine
The result:7
```

该结果代表我们所取样本为数字 7。

以上为神经网络运行的基本方法，但是我们还有很多工作可以做。比如，可以记录所有用户的数据集，这样就可以通过人工标注产生新的训练样本集，将其添加到原来的训练样本集之后。我们在当前网络参数基础上再次进行训练时，可以适当减小学习率，和原来一样，在验证样本集上看是否停止运行，在测试样本集上检查最终效果。这样随着网络的运行，积累的训练样本越来越多，只要网络足够大（有足够多的层数和每层神经元数），网络预测性能就会逐渐提高。

为了方便，我们的预测样本取的是 MNIST 手写数字识别数据集测试样本集中的样本，如果允许用户直接上传 JPG 文件，可以对这些 JPG 文件进行预处理，形成同样格式的样本，只有这样普通用户才有可能使用我们的接口。

15.3 AJAX 请求调用神经网络

我们首先要向用户显示请求图片识别服务的页面，用户可以在这个页面中上传需要识别的图片，这里只是一个示例程序，所以我们要求用户仅上传 JPG 文件，文件为黑底白字，并用分辨率为 28×28、内容为 0~9 的数字。当用户将图片上传完成后，单击识别按钮，系统将自动识别图片上的数字，并显示在识别结果区域。

15.3.1 显示静态网页

我们需要在程序中加入 Web 服务器子系统，在 app_main.py 中加入以下代码：

```
1 import sys
2 sys.path.append('./lib/Theano')
3 sys.path.append('./lib/cherrypy')
4 import os
5 import theano
6 import cherrypy
7 import app_global as ag
8 import model.m_mysql as db
9 import controller.c_mlp as mlp
10 import app_web as app_web
11
12 if __name__ == '__main__':
13     print('starting up...')
14     db.init_db_pool()
15     app_web.startup()
16     db.rdb_pool_cleaner.join()
17     db.wdb_pool_cleaner.join()
```

画线部分的代码是新加入的 Web 服务器入口程序。

还需要定义新的全局文件，代码如下：

```
1 import os
2
3 dataset_dir = os.getcwd() + '/data/'
4 ann_dir = os.getcwd() + '/repository/'
5 web_dir = os.getcwd()
6 upload_dir = os.getcwd() + '/upload/'
```

Web 服务器配置文件的代码如下：

```
1 import sys
2 sys.path.append('./lib/cherrypy')
3 import cherrypy
4 from cherrypy.lib import auth_digest
5
6 g_auth_users = {'yant': '123456'}
7
8 app_conf = {'/': {
9     'tools.staticdir.root': '/home/osboxes/dev/wky/work/book/chp15/',
10    'request.dispatch': cherrypy.dispatch.MethodDispatcher(),
11 },
12 '/web/public': {
13     'tools.staticdir.on': True,
14     'tools.staticdir.dir': 'public',
15 },
16 '/api': {
17     'tools.auth_digest.on': True,
18     'tools.auth_digest.realm': '192.168.1.16',
19     'tools.auth_digest.get_ha1': auth_digest.get_ha1_dict_plain(g_auth_u
20 sers),
21     'tools.auth_digest.key': 'a565c27146791c9b'
22 }
```

第 6 行：定义 HTTP Digest 认证时的用户列表。

第 10 行：定义 request.dispatch 为 MethodDispatcher，所以我们可以定义 REST 风格的 API。

第 12~15 行：定义 public 目录下的内容为静态内容。

第 16~21 行：定义 api 目录下的内容需要进行 HTTP Digest 认证。

下面来看 Web 服务器入口程序，代码如下：

```
1 import sys
2 sys.path.append('./lib/cherrypy')
3 import cherrypy
4 import json
5 import conf.web_conf as web_conf
6 from controller.page_controller import Page_Controller
7 from controller.ajax_controller import Ajax_Controller
8 from controller.web_controller import Web_Controller
9 from controller.upload_controller import File_Uploader
10
11 def startup():
12     cherrypy.config.update({
13         'server.socket_host': '192.168.1.16',
14         'server.socket_port': 8090,
15     })
16     page_controller = Page_Controller()
17     ajax_controller = Ajax_Controller()
18     web_controller = Web_Controller()
19     upload_controller = File_Uploader()
20     cherrypy.tree.mount(page_controller, '/web/pages', web_conf.app_conf)
21     cherrypy.tree.mount(web_controller, '/', web_conf.app_conf)
22     cherrypy.tree.mount(ajax_controller, '/wky/dl/mlp', web_conf.app_conf)
23     cherrypy.tree.mount(upload_controller, '/upload', {'/upload': {}})
24     cherrypy.engine.start()
25     cherrypy.engine.block()
```

第 6 行：引入用于显示页面的控制器类。

第 7 行：引入用于提供 RESTful AJAX API 的控制器类。

第 8 行：引入用于服务静态内容的控制器。

第 9 行：引入用于处理文件上传的控制器。

第 11 行：定义 Web 服务器启动函数。

第 12~15 行：监听本地 IP 地址的 8090 端口。

第 16 行：生成页面显示控制器对象 `page_controller`。

第 17 行：生成 RESTful AJAX API 控制器对象 `ajax_controller`。

第 18 行：生成静态内容显示控制器对象 `web_controller`。

第 19 行：生成文件上传控制器对象 `upload_controller`。

第 20 行：以指定配置加载页面显示控制器对象。

第 21 行：以指定配置加载 RESTful AJAX API 控制器对象。

第 22 行：以指定参数加载静态内容显示控制器对象。

第 23 行：以指定参数加载文件上传控制器对象。

第 24、25 行：启动 Web 服务器。

上面的代码将我们用到的控制器全部加载了，现在只需要看页面显示控制器 `Page_Controller`，代码如下：

```
1 import sys
2 sys.path.append('./lib/cherrypy')
3 import cherrypy
4 import app_global as ag
5
6 class Page_Controller(object):
7     exposed = True
8     def __init__(self):
9         self.web_dir = ag.web_dir
10
11     def GET(self, params={}):
12         cmd = params['kwargs']['cmd']
13         func = getattr(self, cmd)
14         return func(params)
15
16     def read_html(self, file):
17         fo = open(self.web_dir + file, 'r')
18         try:
19             html = fo.read()
20         finally:
21             fo.close()
22         return html
23
24     def show_rest01(self, params):
25         return self.read_html('/public/rest01.html')
26
27     def show_mlp_1(self, params):
28         return self.read_html('/public/mlp_1.html')
```

第 7 行：由于我们要提供 RESTful 格式的 URL，因此需要加入此行。

第 11 行：定义 GET 请求处理函数。我们访问每个网页实际上都是发送 HTTP GET 请求。

第 12 行：取出命令参数 `cmd`，一般形式为 `show_***`，表示显示哪个页面。

第 13 行：从本类中找出同名的方法。

第 14 行：调用该方法。

第 16~22 行：定义读取 HTML 文件内容的方法，参数为 HTML 文件名（不加路径）。

第 27、28 行：定义显示服务请求页面 `public/mlp_1.html`。其实这里可以引入模板技术，

通过参数来生成动态页面内容。

下面来看看用户申请图片识别服务的界面，我们在这里使用了 jQuery FileUpload 控件来完成图片文件的上传，需要先下载该控件。进入 chp15/public 目录，运行以下命令：

```
git clone https://github.com/blueimp/jQuery-File-Upload.git jqfu
```

进入 chp15/public/jqfu/js 目录，下载两个 JS 文件，代码如下：

```
wget http://blueimp.github.io/JavaScript-Load-Image/js/load-image.all.min.js
wget http://blueimp.github.io/JavaScript-Canvas-to-Blob/js/canvas-to-blob.min.js
```

下面来看看服务请求页面，页面内容如图 15.4 所示。



图 15.4 服务请求页面的内容

选择要上传的图片文件，如图 15.5 所示。



图 15.5 要上传的图片文件

图片上传结束后，单击“图像识别”按钮，如图 15.6 所示。



图 15.6 单击“图像识别”按钮

图片识别结果如图 15.7 所示。



图 15.7 图片识别结果

图片识别服务请求页面的代码如下：

```
1 <!DOCTYPE HTML>
2 <html lang="en">
3 <head>
4 <!-- Force latest IE rendering engine or ChromeFrame if installed -->
5 <!--[if IE]><meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1"><![endif]-->
6 <meta charset="utf-8">
7 <title>jQuery File Upload Demo - Basic Plus version</title>
8 <meta name="description" content="File Upload">
9 <meta name="viewport" content="width=device-width, initial-scale=1.0">
10 <!-- Bootstrap styles -->
11 <link rel="stylesheet" href="public/css/bootstrap.min.css">
12 <!-- Generic page styles -->
13 <link rel="stylesheet" href="public/jqfu/css/style.css">
14 <!-- CSS to style the file input field as button and adjust the Bootstrap progress bars -->
15 <link rel="stylesheet" href="public/jqfu/css/jquery.fileupload.css">
16 </head>
17 <body>
18 <div class="navbar navbar-default navbar-fixed-top">
19   <div class="container">
20     <div class="navbar-header">
21       <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-fixed-top .navbar-collapse">
```

```

22         <span class="icon-bar"></span>
23         <span class="icon-bar"></span>
24         <span class="icon-bar"></span>
25     </button>
26     <a class="navbar-brand" href="https://github.com/blueimp/jquery-
File-Upload">深度学习框架</a>
27 </div>
28 <div class="navbar-collapse collapse">
29     <ul class="nav navbar-nav">
30         <li><a href="https://github.com/blueimp/jquery-File-Upload/t
ags">开源项目</a></li>
31         <li><a href="https://github.com/blueimp/jquery-File-Upload">
博客</a></li>
32         <li><a href="https://github.com/blueimp/jquery-File-Upload/w
iki">电子书</a></li>
33         <li><a href="https://blueimp.net">论坛</a></li>
34     </ul>
35 </div>
36 </div>
37 </div>
38 <div class="container">
39     <h1>多层感知器模型</h1>
40     <h2 class="lead">MNIST手写数字识别</h2>
41     <!-- The fileinput-button span is used to style the file input field as
button -->
42     <span id="add_file_span" class="btn btn-success fileinput-button">
43         <i class="glyphicon glyphicon-plus"></i>
44         <span>添加文件</span>
45         <!-- The file input field used as target for the file upload widget
-->
46         <input id="fileupload" type="file" name="files[]" multiple>
47     </span>
48     <br>
49     <br>
50     <!-- The global progress bar -->
51     <div id="progress" class="progress">
52         <div class="progress-bar progress-bar-success"></div>
53     </div>
54     <!-- The container for the uploaded files -->
55     <div id="files" class="files"></div>
56     <div id="upload_notes" class="panel panel-default">
57         <div class="panel-heading">
58             <h3 class="panel-title">小提示</h3>
59         </div>
60         <div class="panel-body">
61             <ul>
62                 <li>仅支持JPG文件</li>
63                 <li>图片分辨率28*28，黑底白字</li>
64                 <li>内容为0~9数字</li>
65             </ul>
66         </div>
67     </div>
68     <span id="classify_btn" class="btn btn-success fileinput-button" style="
display: none;">
69         <i class="glyphicon glyphicon-plus"></i>
70         <span>图像识别</span>
71     </span>
72     <br /><br />
73     <blockquote id="classify_result" style="display: none;">
74         <p>
75             识别结果：<span id="result">***</span>
76         </p>
77     </blockquote>
78     <input type="hidden" id="samples_id" />
79 </div>
80 <script src="public/js/jquery-3.1.1.js"></script>
81 <!-- The jQuery UI widget factory, can be omitted if jQuery UI is already in
cluded -->
82 <script src="public/jqfu/js/vendor/jquery.ui.widget.js"></script>
83 <!-- The Load Image plugin is included for the preview images and image resi
zing functionality -->
84 <script src="public/jqfu/js/load-image.all.min.js"></script>
85 <!-- The Canvas to Blob plugin is included for image resizing functionality
-->
86 <script src="public/jqfu/js/canvas-to-blob.min.js"></script>

```

```

87 <!-- Bootstrap JS is not required, but included for the responsive demo navigation -->
88 <script src="public/js/bootstrap.min.js"></script>
89 <!-- The Iframe Transport is required for browsers without support for XHR file uploads -->
90 <script src="public/jqfu/js/jquery.iframe-transport.js"></script>
91 <!-- The basic File Upload plugin -->
92 <script src="public/jqfu/js/jquery.fileupload.js"></script>
93 <!-- The File Upload processing plugin -->
94 <script src="public/jqfu/js/jquery.fileupload-process.js"></script>
95 <!-- The File Upload image preview & resize plugin -->
96 <script src="public/jqfu/js/jquery.fileupload-image.js"></script>
97 <!-- The File Upload audio preview plugin -->
98 <script src="public/jqfu/js/jquery.fileupload-audio.js"></script>
99 <!-- The File Upload video preview plugin -->
100 <script src="public/jqfu/js/jquery.fileupload-video.js"></script>
101 <!-- The File Upload validation plugin -->
102 <script src="public/jqfu/js/jquery.fileupload-validate.js"></script>
103 <script>
104 /*jslint unparam: true, regexp: true */
105 /*global window, $ */
106 $(function () {
107     'use strict';
108     $("#classify_btn").bind("click", classify_img);
109     // Change this to the location of your server-side upload handler:
110     var url = '/upload/upload_mlp1',
111         uploadButton = $('<button/>')
112             .addClass('btn btn-primary')
113             .prop('disabled', true)
114             .text('上传中...')
115             .on('click', function () {
116                 var $this = $(this),
117                     data = $this.data();
118                 $this
119                     .off('click')
120                     .text('取消')
121                     .on('click', function () {
122                         $this.remove();
123                         data.abort();
124                     });
125                 data.submit().always(function () {
126                     $this.remove();
127                 });
128             });
129     $('#fileupload').fileupload({
130         url: url,
131         dataType: 'json',
132         autoUpload: false,
133         acceptFileTypes: /(\\.|\/)(gif|jpe?g|png)$/i,
134         maxFileSize: 999000,
135         // Enable image resizing, except for Android and Opera,
136         // which actually support image resizing, but fail to
137         // send Blob objects via XHR requests:
138         disableImageResize: /Android(?!.*Chrome)|Opera/
139             .test(window.navigator.userAgent),
140         previewMaxWidth: 100,
141         previewMaxHeight: 100,
142         previewCrop: true
143     }).on('fileuploadadd', function (e, data) {
144         data.context = $('<div/>').appendTo('#files');
145         $.each(data.files, function (index, file) {
146             var node = $('<p/>')
147                 .append($('<span/>').text(file.name));
148             if (!index) {
149                 node
150                     .append('<br>')
151                     .append(uploadButton.clone(true).data(data));
152             }
153             node.appendTo(data.context);
154         });
155     }).on('fileuploadprocessalways', function (e, data) {
156         var index = data.index,
157             file = data.files[index],
158             node = $(data.context.children()[index]);
159         if (file.preview) {

```

```

160         node
161         .prepend('<br>')
162         .prepend(file.preview);
163     }
164     if (file.error) {
165         node
166         .append('<br>')
167         .append($('<span class="text-danger">').text(file.error));
168     }
169     if (index + 1 === data.files.length) {
170         data.context.find('button')
171         .text('上传文件')
172         .prop('disabled', !!data.files.error);
173     }
174     }).on('fileuploadprogressall', function (e, data) {
175         var progress = parseInt(data.loaded / data.total * 100, 10);
176         $('#progress .progress-bar').css(
177             'width',
178             progress + '%'
179         );
180     }).on('fileuploaddone', function (e, data) {
181         $('#samples_id').val(data.result.samples_id);
182         $('#add_file_span').css("display", "none");
183         $('#progress').css("display", "none");
184         $('#upload_notes').css("display", "none");
185         $('#classify_btn').css("display", "block");
186         $('#classify_btn').width(100);
187         $.each(data.result.files, function (index, file) {
188             if (file.url) {
189                 var link = $('<a>')
190                 .attr('target', '_blank')
191                 .prop('href', file.url);
192                 $(data.context.children()[index])
193                 .wrap(link);
194             } else if (file.error) {
195                 var error = $('<span class="text-danger">').text(file.error);
196                 $(data.context.children()[index])
197                 .append('<br>')
198                 .append(error);
199             }
200         });
201     }).on('fileuploadfail', function (e, data) {
202         $.each(data.files, function (index) {
203             var error = $('<span class="text-danger">').text('File upload f
204             iled.');
```

```

232     alert("图像识别失败：" + JSON.stringify(msg) + "！");
233 }
234 </script>
235 </body>
236 </html>

```

这段代码的主体是用 Bootstrap 写的 HTML 页面，这里就不一一讲解了，我们只对与图片上传相关的内容进行描述，重点在于讲解怎样通过 AJAX 服务调用图片识别服务，并显示识别结果的 JS 代码。

第 13 行：引入 jQuery FileUpload 控件格式文件。

第 15 行：引入 jQuery FileUpload 控件 jQuery 格式文件。

第 42~47 行：定义添加文件按钮，其中包括文件上传控件。

第 51~53 行：定义进度条，上传文件时显示进度情况。

第 56~67 行：定义文件上传提示信息，提示用户输入正确内容。由于这里是演示程序，所以没进行数据合法性检查。

第 68~71 行：定义识别图像按钮，初始时其是隐藏的，只有在用户完成上传图像后，其才会显示出来。

第 73~77 行：图片识别结果，初始时其是隐藏的，只有单击识别图像后，其才显示图片识别结果。

第 78 行：样本集编号，当上传图片文件成功后，在上传成功消息处理函数中，将服务器返回的样本集编号填写到该文本框中。

第 106 行：定义页面入口函数。

第 108 行：为识别图像按钮绑定点击事件 `classify_img`，通过 AJAX 请求向服务器发送图片识别请求。

jQuery FileUpload 的逻辑是比较复杂的，不是我们要讲解的重点，这里就不详细介绍其代码了。对于我们目前的任务而言，只需要关心上传文件成功的消息响应函数，并在这个函数中加入我们的代码即可。找到 `fileuploaddone`，添加第 180~200 行代码。

第 181 行：取出服务器返回的上传图片所对应的样本集编号，并将该值赋给 ID 为 `samples_id` 的输入框。

第 182 行：隐藏添加文件按钮。

第 183 行：隐藏文件上传进度条按钮。

第 184 行：隐藏文件上传提示信息。

第 185 行：显示识别图片按钮。

第 186 行：将识别图片按钮的宽度设置为 100px。

第 212 行：定义识别图片按钮的消息响应函数 `classify_img`。

第 213 行：定义神经网络编号 `ann_id`。

第 214 行：取出刚才从服务器得到的上传图片所对应的样本集编号 `samples_id`。

第 215 行：构造图像识别 URL。

第 216~222 行：通过 AJAX GET 请求，调用图像识别服务。

第 224 行：定义图像识别服务成功消息响应函数。

第 225 行：显示图像识别结果显示区域。

第 227 行：取出返回的图像识别结果并解析为 JSON 对象。

第 228 行：取出具体的识别结果。

第 229 行：将识别结果显示在识别结果显示区域内。

第 231～233 行：定义图像识别服务失败消息响应函数，仅显示出错信息。

15.3.2 上传图片文件

从网页中可以看出，图片上传时调用的 URL 为 `upload/upload_mlp1`，所以需要在 Web 服务器端定义 `upload_controller` 来处理这一请求，代码如下：

```
1 import sys
2 sys.path.append('./lib/cherrypy')
3 import cherrypy
4 import cgi
5 import tempfile
6 import os
7 import shutil
8 import time
9 from PIL import Image
10 import pickle
11 import numpy as np
12 import app_global as ag
13
14 class Field_Storage(cgi.FieldStorage):
15     def make_file(self, binary=None):
16         return tempfile.NamedTemporaryFile()
17
18 def no_body_process():
19     cherrypy.request.process_request_body = False
20
21 cherrypy.tools.noBodyProcess = cherrypy.Tool('before_request_body', \
22                                             no_body_process)
23 cherrypy.server.socket_timeout = 60
24 cherrypy.server.max_request_body_size = 0
25
26 class File_Uploader(object):
27     @cherrypy.expose
28     def index(self, params={}):
29         return 'File_Uploader'
30
31     @cherrypy.expose
32     @cherrypy.tools.noBodyProcess()
33     @cherrypy.tools.json_out()
34     def upload_mlp1(self, data_file=None):
35         cherrypy.response.timeout = 3600
36         req_headers = {}
37         for hdr in cherrypy.request.headers.items():
38             req_headers[hdr[0].lower()] = hdr[1]
39         form_fields = Field_Storage(fp=cherrypy.request.rfile, \
40                                   headers = req_headers, environ = {\
41                                       'REQUEST_METHOD': 'POST'}, \
42                                       keep_blank_values=True)
43         data_file = form_fields['files[]']
44         src_file = data_file.file.name
45         uploaded_file = 'f_' + \
46             str(time.time()).replace('.', '_') + \
47             os.path.splitext(data_file.filename)[1]
48         dest_file = ag.upload_dir + uploaded_file
49         shutil.copy(src_file, dest_file)
50         samples_id = self.save_samples(dest_file)
51         resp = {'status': 'Ok', 'samples_id': samples_id, 'url': 'aaa'}
52         return resp
53
```

```

54     def save_samples(self, img_file):
55         im = np.array(Image.open(img_file).convert('L'), 'f')
56         sample_data = np.arange(28*28)
57         for i in range(28):
58             for j in range(28):
59                 if im[i][j]<128:
60                     sample_data[(i-1)*28+j] = 0.0
61                 else:
62                     sample_data[(i-1)*28+j] = 1.0
63         samples_data = np.array([sample_data])
64         import model.m_samples as m_samples
65         user_id = 1
66         samples_id = m_samples.add_samples(user_id)
67         samples_file = open(ag.dataset_dir + 's_' + str(samples_id) \
68                             + '.pkl', 'wb')
69         pickle.dump(samples_data, samples_file)
70         samples_file.close()
71         return samples_id

```

第 4~7 行：引入文件上传相关库。

第 9 行：引入读取图片文件内容库。

第 10 行：引入序列化库，用于保存图片转为样本时的数据。

第 14~16 行：定义 `cgi.FieldStorage` 子类，用于获取命名的上传文件。

第 18~22 行：定义不处理消息体。

第 23 行：定义请求超时时间。

第 24 行：定义上传文件大小限制。

第 26 行：定义文件上传类。

第 31~34 行：定义接口 `upload_mlp1`，不处理消息体，返回值为 JSON 格式。

第 36~38 行：求出请求的 Headers。

第 39~42 行：求出窗体 form 中的所有内容，包括普通输入框和文件上传控件。

第 43 行：获取文件上传控件内容 `data_file`。

第 44 行：上传成功的临时文件全路径文件名。

第 45~47 行：生成上传文件最终的文件名，格式为 `f_+当前时间+上传文件的扩展名`。

第 48 行：生成最终上传文件的全路径文件名。

第 49 行：将上传的文件从临时位置复制到正式位置。

第 50 行：读取图片文件内容，生成多层感知器模型所需样本集格式，将其保存到样本集文件中，返回样本集编号。

第 51、52 行：生成响应 JSON 字符串并返回。

第 54 行：定义将图片文件转化为多层感知器样本集格式的函数。

第 55 行：以灰度图像方式读出图片每像素的内容。

第 56 行：生成数组 `sample_data` 保存单个样本数据。

第 57、58 行：对每个像素点进行循环。

第 59~62 行：如果像素点值小于 128，则最终样本点值为 0；如果像素点值大于或等于 128，则最终样本点值为 1。

第 63 行：将其包装为二维数组。

第 64 行：引入样本集模型类。

第 65 行：定义 `user_id` 为一个实验值，在下一节中，我们将在验证请求合法性中讨论如何获取 `user_id` 的值。

第 66 行：将样本集保存到数据库中。

第 67 行：打开样本集文件。

第 68、69 行：将样本集数据保存到样本集文件中。

第 70 行：关闭样本集文件。

第 71 行：返回样本集编号。

下面来看看样本集模型类，代码如下：

```
1 import model.m_mysql as db
2
3 def add_samples(user_id):
4     sql = 'insert into t_samples(user_id, create_date, labeled) \
5           values(%s, sysdate(), 0)'
6     params = (user_id)
7     samples_id, affected_rows = db.insert(sql, params)
8     return samples_id
```

第 3 行：定义向样本集数据表添加记录的函数。

第 4、5 行：定义 `insert` 语句，用 `%s` 代表参数占位符。

第 6 行：定义 `insert` 语句中的参数值。

第 7 行：调用 `m_mysql` 模块中的 `insert` 函数，返回刚插入数据库表中的主键值和影响行数。

第 8 行：返回主键值。

数据库插入操作实现的代码如下：

```
103 def insert(sql, params):
104     conn_obj = get_wdb_connection()
105     conn = conn_obj['conn']
106     result = insert_t(conn, sql, params)
107     close_db_connection(conn_obj)
108     return result
109
110 def insert_t(conn, sql, params):
111     cursor = conn.cursor()
112     affected_rows = cursor.execute(sql, params)
113     conn.commit()
114     cursor.close()
115     pk = cursor.lastrowid
116     return (pk, affected_rows)
```

第 103 行：定义不在数据库事务内的数据添加函数，`sql` 为带参数的 SQL 语句，`params` 为参数的具体值。

第 104 行：从写数据库连接池获取数据连接对象。

第 105 行：从数据库连接对象中取出数据库连接。

第 106 行：调用带有数据库连接参数的数据库添加函数 `insert_t`。

第 107 行：关闭数据库连接对象，实际为将数据库连接放回写数据库连接池。

第 108 行：返回数据库添加操作结果，包括刚插入数据库记录的主键值和影响行数。

第 110 行：定义数据库添加函数，`conn` 表示数据库连接，`sql` 为带参数的 SQL 语句，`params` 为 SQL 语句中的参数值。

第 111 行：从数据库连接中生成数据库操作所需的游标。

第 112 行：在游标上执行 SQL 语句，参数为 SQL 语句及其参数值，返回值为影响行数。

第 113 行：将修改同步到数据库中。

第 114 行：关闭游标。

第 115 行：获取刚插入记录的主键。

第 116 行：返回主键值和影响行数。

15.3.3 AJAX 接口

当用户单击识别图像按钮后，就会调用 RESTful AJAX API 接口：

```
1 import sys
2 sys.path.append('./lib/cherrypy')
3 import cherrypy
4 import json
5 import conf.web_conf as web_conf
6 import app_global as ag
7
8 class Ajax_Controller(object):
9     exposed = True
10     def __init__(self):
11         self.web_dir = ag.web_dir
12
13     @cherrypy.tools.json_out()
14     def GET(self, params={}):
15         return self.http_method(params)
16
17     def http_method(self, params):
18         cmd = params['kwargs'].get('cmd', 'unknown')
19         cls = params['kwargs'].get('cls', self)
20         fullname = 'controller.' + cls
21         mdl = __import__(fullname)
22         print(mdl)
23         obj = sys.modules[fullname]
24         func = getattr(obj, cmd)
25         return func(params)
```

第 13、14 行：定义 HTTP GET 请求处理函数，调用本类的 `http_method` 方法。

第 17 行：定义 `http_method` 方法。

第 18 行：求出 `cmd` 参数的值。

第 19 行：求出 `cls` 参数的值。

第 20 行：生成全路径的类名。

第 21 行：引入控制器类。

第 23 行：取出刚刚引入的类对象。

第 24 行：在该类对象中查找与 `cmd` 同名的方法。

第 25 行：调用该方法。

在这里我们将调用 `controller/c_mlp.py` 模块，代码如下：

```
1 import model.m_ann_hyper_param as m_ahp
2 import model.m_ann_version as m_av
3 from ann.mlp.mlp_mnist_engine import MlpMnistEngine
4 import app_global as ag
5
6 def get_ahps(ann_id):
7     raw_ahps = m_ahp.get_ann_hyper_params(ann_id)
8     ahps = {}
9     ahps['learning_rate'] = float(raw_ahps['1'])
```

```

10     ahps['layers'] = int(raw_ahps['2'])
11     ahps['layer_nums'] = raw_ahps['3']
12     ahps['momentum'] = float(raw_ahps['4'])
13     ahps['L1'] = float(raw_ahps['5'])
14     ahps['L2'] = float(raw_ahps['6'])
15     ahps['batch_size'] = int(raw_ahps['7'])
16     ahps['n_epochis'] = int(raw_ahps['8'])
17     return ahps
18
19 def get_ann_version(ann_id):
20     return m_av.get_ann_version(ann_id)
21
22 g_mlp_engines = {}
23 def create_mlp_engine(ann_id):
24     if str(ann_id) in g_mlp_engines.keys():
25         print('#### return old engine')
26         g_mlp_engines[str(ann_id)]
27     ahps = get_ahps(ann_id)
28     ann_version_info = get_ann_version(ann_id)
29     ahps['ann_version_id'] = ann_version_info['ann_version_id']
30     ahps['dataset'] = 'mnist.pkl.gz'
31     g_mlp_engines[str(ann_id)] = MlpMnistEngine(ahps)
32     return g_mlp_engines[str(ann_id)]
33
34 def train(ann_id):
35     mlp_engine = create_mlp_engine(ann_id)
36     mlp_engine.train()
37
38 def run(ann_id, samples):
39     mlp_engine = create_mlp_engine(ann_id)
40     return mlp_engine.predict(samples)
41
42 def classify_img(params):
43     kwargs = params['kwargs']
44     args = params['args']
45     ann_id = kwargs['ann_id']
46     samples_id = kwargs['samples_id']
47     samples = get_samples(samples_id)
48     result = run(ann_id, samples)
49     resp = {'status': 'Ok', 'result': '%s' % result}
50     return resp
51
52 import pickle
53 def get_samples(samples_id):
54     samples_file = open(ag.dataset_dir + 's_' + samples_id + '.pkl', 'rb')
55     samples_data = pickle.load(samples_file)
56     samples_file.close()
57     return samples_data

```

这个类的部分代码在前面已经解析过，但是这个类非常重要，代表了怎样使用多层感知器模型，所以把完整的代码列在这里。

第 22 行：定义了一个全局字典，其中保存神经网络编号和多层感知器模型对象的对应关系，这样就可以只创建一次多层感知器引擎类实例了。

第 23 行：定义创建多层感知器引擎类对象的方法。

第 24~26 行：如果该神经网络编号对应的多层感知器模型类在全局变量 `g_mlp_engines` 中有实例，则直接返回该实例。

第 27 行：如果之前没有创建过，则获取该神经网络编号对应的超参数信息。

第 28 行：获取神经网络编号当前的最新版本信息。

第 29 行：指定需要创建的版本。

第 30 行：指定训练用数据集文件名。

第 31、32 行：创建多层感知器引擎。

第 34 行：定义多层感知器训练方法。

第 35 行：创建多层感知器引擎类实例。

第 36 行：调用多层感知器引擎的训练方法 `train`。

第 38 行：定义多层感知器运行方法。

第 39 行：创建多层感知器引擎类实例。

第 40 行：调用多层感知器引擎的预测方法 `predict`。

第 42 行：定义识别图像方法，当用户在网页上单击识别图像按钮时调用该方法。

第 43 行：获取 `QueryString` 中的参数字典。

第 44 行：获取 URL 路径上的参数元组。

第 45 行：从 `QueryString` 参数字典中取出神经网络编号 `ann_id`。

第 46 行：从 `QueryString` 参数字典中取出样本集编号 `samples_id`。

第 47 行：获取样本集编号对应的样本集数据。

第 48 行：调用本类的 `run` 方法。

第 49、50 行：生成返回值 JSON 字符串并返回。

第 52 行：引入 Python 序列化类。

第 53 行：定义获取样本集数据方法，参数为样本集编号。

第 54 行：根据样本集 ID 形成样本集文件名，并打开该文件。

第 55 行：通过序列化类 `pickle.load` 载入样本集数据。

第 56 行：关闭样本集文件。

第 57 行：返回样本集数据。

下面来看多层感知器引擎类的预测方法，代码如下：

```
181     def predict(self, samples):
182         classifier = pickle.load(open(ag.ann_mf_dir +
183                                     self.model_file, 'rb'))
184         predict_model = theano.function(
185             inputs = [classifier.input],
186             outputs = classifier.logRegressionLayer.y_pred
187         )
188         return predict_model(samples)
```

第 182、183 行：从训练好的模型文件中读取参数和超参数信息，生成分类器 `classifier`，这个分类器由输入层、隐藏层、逻辑回归层及输出层组成。

第 184~187 行：定义 `theano` 函数 `predict_model`，输入为分类器的输入，输出为分类器逻辑回归层的输出标签集。

第 188 行：以样本集数据为参数，调用 `theano` 函数 `predict_model`，并返回输出标签集。

15.4 请求合法性验证

到目前为止，我们已经向读者演示了一个极简的深度学习服务云平台的开发过程。但是我们省略了一个非常重要的问题，即安全问题。如果我们的服务要放到公网上，就有可能被人恶意攻击。最简单的一种攻击方式就是直接复制我们的请求 URL，并采用多线程方

式不停地向我们的服务器发送请求，因为我们的服务器有数据库操作、深度学习网络前向运算，这些都是耗时又耗资源的操作，很容易就被别人攻击瘫痪，这就是所谓的拒绝服务（Denial of Service, DOS）攻击，所以我们应该避免成为这种攻击的受害者。

由于本书的重点是与深度学习算法实践相关的内容，Web 服务器如何进行黑客攻防不是本书的重点，所以只简单地向读者介绍一下怎样预防这种最简单的攻击形式，正式上线系统与黑客做斗争是一个长期持续的过程，是一个与黑客攻击手段相互适应的过程。但是基本技术原理与我们介绍的类似，读者可以举一反三。

为了防止拒绝服务攻击，我们可以采用请求认证的方式。首先，规定只有系统的用户才能发送服务请求；其次，所有用户与服务器端分享一个密钥，该密钥会定期更新；最后，在每个请求后面加一个 `mac` 参数，其值为请求共享密钥中用户编号加上请求编号，并做 SHA1 摘要得到的字符串。这样，服务器收到每个请求后，都会通过对请求 URL 加上共享密钥做 SHA1 摘要，与请求中的 `mac` 参数进行比较，如果一样则认为是合法请求。

另外，我们还规定，用户每次发送请求时必须生成唯一的请求编号，建议是自增的序号，如果服务器端收到两个相同的请求序号，服务器端就不会对第二个请求进行服务，以此来避免有人中途截获请求，并用同样的请求轰炸我们的服务器。

根据以上原理，我们需要做以下工作：

- (1) 用户注册。
- (2) 用户登录。
- (3) 获取共享密钥。
- (4) 网页端生成请求编号和 SHA1 摘要。
- (5) 服务器端验证 SHA1 摘要和请求编号。

15.4.1 用户注册和登录

下面我们还以多层感知器图像识别为例，来看看怎样对请求进行合法性验证。服务请求页面首先会从 `localStorage` 中获取 `user_id` 和 `shared_key`，如果没有，则显示登录页面要求用户输入登录名和密码，还会显示注册链接。当第一次使用时，用户需要输入姓名、身份证号、登录名、密码进行注册，注册成功后系统将返回给用户 `user_id` 和 `shared_key`，用户可以将这些信息保存到 `localStorage` 中，下次访问时直接使用。

下面来显示服务请求页面，在页面加载时，从 `localStorage` 中读取 `user_id` 和 `shared_key`，如果没有读到该信息，则显示登录界面。由于网页内容较多，这里只列出与上节相比需要修改的部分，代码如下，其他内容读者可以参考上节内容。

```
38 <div class="container">
39   <h1>多层感知器模型（安全验证）</h1>
40   <h2 class="lead">MNIST手写数字识别</h2>
41   <div id="user_div" style="display: none;">
42     欢迎：<span id="user_name_span"></span>! <br />
43     <input type="button" id="logout_btn" value="退出" />
44     <br /><br />
45   </div>
46   <div id="login_div" style="display: none;">
```

```

47     请登录<br />
48     <input type="text" id="login_name" placeholder="登录名" /><br />
49     <input type="password" id="login_pwd" placeholder="密码" /><br />
50     <input type="button" id="login_btn" value="登录" /><br />
51     <a id="reg_span" href="#">我是新用户，请注册</a>
52 </div>
53 <div id="reg_div" style="display: none;">
54     注册新用户<br />
55     <input type="text" id="user_name" placeholder="姓名" /><br />
56     <input type="text" id="email" placeholder="邮件" /><br />
57     <input type="text" id="r_login_name" placeholder="登录名" /><br />
58     <input type="password" id="r_login_pwd" placeholder="密码" /><br />
59     <input type="button" id="reg_btn" value="注册" /><br />
60     <a id="login_span" href="#">登录</a>
61 </div>
62 <!-- The fileinput-button span is used to style the file input field as
    button -->
63 <span id="add_file_span" class="btn btn-success fileinput-button">
64     <i class="glyphicon glyphicon-plus"></i>
65     <span>添加文件</span>
66     <!-- The file input field used as target for the file upload widget
    -->
67     <input id="fileupload" type="file" name="files[]" multiple>
68 </span>

```

第 41~45 行：当用户为登录状态时，显示用户的信息（用户名），并显示退出登录按钮。

第 46~52 行：如果用户没有登录，则显示登录界面。单击“我是新用户，请注册”按钮可以进入用户注册界面。

第 53~61 行：用户注册界面，用于新用户注册。

下面是用户注册、登录、退出所对应的 JavaScript 程序：

```

127 $(function () {
128     'use strict';
129     $("#reg_span").bind("click", function() {
130         $("#reg_div").css("display", "block");
131         $("#login_div").css("display", "none");
132     });
133     $("#login_span").bind("click", function() {
134         $("#reg_div").css("display", "none");
135         $("#login_div").css("display", "block");
136     });
137     $("#login_btn").bind("click", login_user);
138     $("#reg_btn").bind("click", reg_user);
139     $("#logout_btn").bind("click", logout_user);
140     var user_id = localStorage.getItem("user_id");
141     if (!user_id || user_id == "") {
142         $("#login_div").css("display", "block");
143         $("#add_file_span").css("display", "none");
144         $("#progress").css("display", "none");
145         $("#upload_notes").css("display", "none");
146     } else {
147         $("#user_div").css("display", "block");
148         $("#user_name_span").text(localStorage.getItem("user_name"));
149         $("#add_file_span").css("display", "block");
150         $("#add_file_span").width(100);
151         $("#progress").css("display", "block");
152         $("#upload_notes").css("display", "block");
153     }

```

第 129~132 行：单击用户登录界面下方的“我是新用户，请注册”按钮，显示用户注册 div，同时隐藏用户登录 div。

第 133~136 行：单击用户注册界面下方的“登录”按钮，显示用户登录 div，同时隐藏用户注册 div。

第 137 行：定义“登录”按钮单击响应函数。

第 138 行：定义“注册”按钮单击响应函数。

第 139 行：定义“退出”按钮单击响应函数。

第 140 行：从 localStorage 中求出 user_id。

第 141~145 行：当 user_id 为空时，显示用户登录界面，隐藏文件上传中的“添加文件”按钮、文件上传进度条、上传提示信息。

第 146~153 行：否则显示当前用户信息，从 localStorage 中读出用户名显示到界面中，并显示文件上传相关控件。

下面是与用户注册相关的函数，代码如下：

```
317 function reg_user() {
318     var reqUrl = "/wky/dl?cls=c_user&cmd=reg_user";
319     var data = new Object();
320     data.cmd = "reg_user"
321     data.user_name = $("#user_name").val();
322     data.email = $("#email").val();
323     data.login_name = $("#r_login_name").val();
324     data.login_pwd = $("#r_login_pwd").val();
325     $.ajax({
326         url: reqUrl,
327         type: "POST",
328         dataType: "json",
329         data: {
330             json_str: JSON.stringify(data)
331         },
332         success: on_reg_user_ok,
333         error: on_reg_user_error
334     });
335 }
336 function on_reg_user_ok(json) {
337     var user_id = parseInt(json.user_id);
338     var shared_key = json.shared_key
339     localStorage.setItem("user_id", user_id);
340     localStorage.setItem("user_name", $("#user_name").val());
341     localStorage.setItem("shared_key", shared_key);
342     $("#user_div").css("display", "block");
343     $("#user_name_span").text(localStorage.getItem("user_name"));
344     $("#add_file_span").css("display", "block");
345     $("#progress").css("display", "block");
346     $("#upload_notes").css("display", "block");
347     $("#reg_div").css("display", "none");
348 }
349 function on_reg_user_error(msg) {
350     alert("注册用户失败：" + JSON.stringify(msg) + "！");
351 }
```

第 317 行：定义用户注册函数。

第 318 行：形成 AJAX 请求，调用 ajax_controller.py 模块中定义的方法。

第 319 行：我们将采用 POST 请求，所以需要先定义一个 Object，用来存放请求体的内容。

第 321 行：向 data 中添加用户名 user_name。

第 322 行：向 data 中添加电子邮件 email。

第 323 行：向 data 中添加登录名 login_name。

第 324 行：向 data 中添加登录密码 login_pwd。

第 325~334 行：向服务器发送 AJAX POST 请求。

第 336 行：定义用户注册成功消息响应函数。

第 337 行：从响应中解析出 user_id。

第 338 行：从响应中解析出 shared_key，用于服务器认证客户端发送请求的合法性。

第 339 行：将 user_id 保存到 localStorage 中。

第 340 行：将用户名 user_name 保存到 localStorage 中。

第 341 行：将 shared_key 保存到 localStorage 中。

第 342 行：显示当前用户的信息界面。

第 343 行：从 localStorage 中读出用户名，显示到界面上的用户信息中。

第 344 行：显示上传文件中的“添加文件”按钮。

第 345 行：显示上传文件中的进度条。

第 346 行：显示上传提示信息。

第 347 行：隐藏用户注册界面。

第 349~351 行：用户注册失败消息响应函数，仅显示具体出错信息。

下面来看看用户登录程序，代码如下：

```
282 function login_user() {
283     var reqUrl = "/wky/dl?cls=c_user&cmd=login_user";
284     var data = new Object();
285     data.cmd = "login_user"
286     data.login_name = $("#login_name").val();
287     data.login_pwd = $("#login_pwd").val();
288     $.ajax({
289         url: reqUrl,
290         type: "POST",
291         dataType: "json",
292         data: {
293             json_str: JSON.stringify(data)
294         },
295         success: on_login_user_ok,
296         error: on_login_user_error
297     });
298 }
299 function on_login_user_ok(json) {
300     if ('Ok' != json.status) {
301         alert("登录失败，请重试");
302         return ;
303     }
304     $("#add_file_span").css("display", "block");
305     $("#add_file_span").width(100);
306     $("#progress").css("display", "block");
307     $("#upload_notes").css("display", "block");
308     $("#user_div").css("display", "block");
309     $("#user_name_span").text(json.user_name);
310     localStorage.setItem("user_id", json.user_id);
311     $("#login_div").css("display", "none");
312 }
313 function on_login_user_error(msg) {
314     alert("用户登录失败：" + JSON.stringify(msg) + "！");
315 }
```

第 282 行：定义用户登录函数。

第 283 行：定义 AJAX 请求 URL，调用 ajax_controller.py 模块中 POST 方法。

第 284 行：定义 Javascript Object 对象 data，用于保存需要提交的数据。

第 285 行：定义 cmd 为 login_user。

第 286 行：将登录名 login_name 保存到 data 中。

第 287 行：将密码 login_pwd 保存到 data 中。

第 288~297 行：向服务器发送 AJAX POST 请求。

第 299 行：用户登录成功消息响应函数。

第 300~303 行：用户登录信息不对，则提示“登录失败，请重试”，并退出函数。

第 304、305 行：用户登录信息正确，显示文件上传中的“添加文件”按钮。

第 306 行：显示文件上传进度条。

第 307 行：显示文件上传提示信息。

第 308 行：显示当前用户信息界面。

第 309 行：在当前用户信息界面中显示用户姓名。

第 310 行：将 user_id 保存到 localStorage 中。

第 311 行：隐藏用户登录界面。

第 313~315 行：用户登录失败消息响应函数，这里仅显示出错信息。

下面来看看“退出”按钮单击响应函数，代码如下：

```
353 function logout_user() {
354     $("#add_file_span").css("display", "none");
355     $("#progress").css("display", "none");
356     $("#upload_notes").css("display", "none");
357     $("#login_div").css("display", "block");
358     $("#user_div").css("display", "none");
359     $("#files").css("display", "none");
360     $("#classify_btn").css("display", "none");
361     $("#classify_result").css("display", "none");
362     localStorage.setItem("user_id", "");
363     localStorage.setItem("shared_key", "");
364 }
```

第 353 行：“退出”按钮单击消息响应函数。

第 354 行：显示文件上传中的“添加文件”按钮。

第 355 行：显示文件上传中的进度条。

第 356 行：显示文件上传提示信息。

第 357 行：显示用户登录界面。

第 358 行：隐藏当前用户信息界面。

第 359 行：隐藏已上传文件信息。

第 360 行：隐藏识别图像按钮。

第 361 行：隐藏图像识别结果。

第 362 行：清空 localStorage 中的 user_id 信息。

第 363 行：清空 localStorage 中的 shared_key 信息。

下面来看看 ajax_controller.py 模块中的 POST 请求处理函数，代码如下：

```
17 @cherry.py.tools.json_out()
18 def POST(self, params={}):
19     json_obj = json.loads(params['kwargs']['json_str'])
20     del params['kwargs']['json_str']
21     params['kwargs']['json_obj'] = json_obj
22     params['kwargs']['cmd'] = json_obj['cmd']
23     return self.http_method(params)
24
25 def http_method(self, params):
26     cmd = params['kwargs'].get('cmd', 'unknown')
27     cls = params['kwargs'].get('cls', self)
28     fullname = 'controller.' + cls
29     mdl = __import__(fullname)
30     print(mdl)
31     obj = sys.modules[fullname]
32     func = getattr(obj, cmd)
33     return func(params)
```

第 17 行：定义本函数输出为 JSON 格式。

第 18 行：定义 POST 请求的响应函数。

第 19 行：将 AJAX POST 请求中的参数字符串转变为 JSON 对象 json_obj。

第 20 行：删除 AJAX POST 请求中的参数字符串 json_str。

第 21 行：将 json_obj 添加到参数中。

第 23 行：调用本类的 http_method 方法，从而调用具体的控制器模块。

第 25 行：定义 HTTP 请求通用处理方法 http_method。

第 26 行：从参数中取出 cmd 参数，代表调用的方法名。

第 27 行：从请求参数中取出 cls，代表模块名。

第 28 行：定义模块全名。

第 29 行：引入该模块，采用 Python 反射机制。

第 31 行：创建该模块对象。

第 32 行：获取该模块中与 cmd 同名的函数。

第 33 行：调用该函数并返回。

这里具体的控制器模块是 controller.c_user，代码如下：

```
1 import json
2 import model.m_ann_hyper_param as m_ahp
3 import model.m_ann_version as m_av
4 from ann.mlp.mlp_mnist_engine import MlpMnistEngine
5 import app_global as ag
6 import model.m_user as m_user
7
8 def reg_user(params):
9     print('reg_user:%s' % params)
10    user_vo = params['kwargs']['json_obj']
11    user_id = m_user.add_user(user_vo)
12    resp = {'status': 'Ok'}
13    resp['user_id'] = user_id
14    resp['shared_key'] = 'wky'
15    return resp
16
17 def login_user(params):
18    json_obj = params['kwargs']['json_obj']
19    login_name = json_obj['login_name']
20    login_pwd = json_obj['login_pwd']
21    user_id = m_user.login_user(login_name, login_pwd)
22    resp = {}
23    if user_id > 0:
24        resp['status'] = 'Ok'
25        user_vo = m_user.get_user_vo(user_id)
26        resp['user_id'] = user_id
27        resp['user_name'] = user_vo['user_name']
28        resp['shared_key'] = 'wky'
29    else:
30        resp['status'] = 'Error'
31    return resp
```

第 8 行：定义用户注册函数。

第 10 行：从请求参数中取出 JSON 对象 json_obj，并将其命名为 user_vo。

第 11 行：调用用户模型模块的 add_user 函数，向数据库中添加新用户，并返回新加入用户的 user_id。

第 12 行：生成响应字典变量 resp。

第 13 行：将 user_id 加入到响应中。

第 14 行：将共享密钥 shared_key 加入到响应中，shared_key 用于服务器认证客户端请求的合法性。

第 15 行：返回响应字典变量。

第 17 行：定义用户登录函数。

第 18 行：从请求参数取出 JSON 对象 json_obj。

第 19 行：取出登录名 login_name。

第 20 行：取出登录密码 login_pwd。

第 21 行：调用用户模型模块的 login_user 函数，求出该登录名、密码所对应的 user_id，如果不存在则返回值为 0。

第 22 行：生成响应字典变量。

第 23 行：如果 user_id 大于 0，即存在此登录名和密码对应的用户，则执行第 24~28 行操作。

第 24 行：设置响应状态为成功。

第 25 行：调用用户模型模块的 get_user_vo 函数，获取指定用户的基本信息。

第 26 行：在响应中添加 user_id。

第 27 行：在响应中添加 user_name。

第 28 行：在响应中添加共享密钥 shared_key。

第 29、30 行：如果登录名和密码不正确，设置响应状态为失败。

在这里将共享密钥设置为统一值，实际上可以针对每个用户设置不同的值，还可以设置每隔一段时间就更新密钥的机制。下面我们就不在这个演示程序中实现这些功能了，读者可以根据实际情况来实现这些功能。

下面来看看用户模型模块，代码如下：

```
1 import model.m_mysql as db
2
3 def add_user(user_vo):
4     sql = 'insert into t_user(user_name, email, login_name, login_pwd, \
5         create_date, salary) values(%, %, %, %, sysdate(), 0.0)'
6     params = (user_vo['user_name'], user_vo['email'], \
7         user_vo['login_name'], user_vo['login_pwd'])
8     user_id, affected_rows = db.insert(sql, params)
9     return user_id
10
11 def login_user(login_name, login_pwd):
12     sql = 'select user_id from t_user where login_name=%s and \
13         login_pwd=%s'
14     params = (login_name, login_pwd)
15     rowcount, rows = db.query(sql, params)
16     user_id = 0
17     if rowcount>0:
18         user_id = rows[0][0]
19     return user_id
20
21 def get_user_vo(user_id):
22     sql = 'select user_name, email, login_name, login_pwd from t_user \
23         where user_id=%s'
24     params = (user_id)
25     rowcount, rows = db.query(sql, params)
26     user_vo = {}
27     if rowcount>0:
28         user_vo['user_name'] = rows[0][0]
29         user_vo['email'] = rows[0][1]
30         user_vo['login_name'] = rows[0][2]
31         user_vo['login_pwd'] = rows[0][3]
32     return user_vo
```

第 3 行：定义用户注册时需要调用的 add_user 函数。

第 4 行：定义带有参数的 insert 语句。

第 6、7 行：定义 insert 语句中参数的具体值。

第 8 行：调用 m_mysql 模块的 insert 函数，返回值为新加入用户的 user_id 和插入的记录数。

第 9 行：返回 user_id。

第 11 行：定义用户登录函数，login_name 为登录名，login_pwd 为密码。

第 12、13 行：定义以 login_name 和 login_pwd 为参数的 SQL 查询语句。

第 14 行：给出 SQL 语句中的参数值元组。

第 15 行：调用 m_mysql 模块的查询函数，以 SQL 和参数元组为参数，返回值为本次取回多少条记录（rowcount）和结果集的二维数组（rows）。

第 16 行：定义 user_id 的初始值为 0。

第 17、18 行：如果取回的记录不为空，则取出 user_id 的值。

第 19 行：返回 user_id。

第 21 行：定义获取用户基本信息函数，user_id 为用户编号。

第 22、23 行：以 user_id 为条件从 t_user 表中查出对应记录的 SQL 语句。

第 24 行：定义 SQL 语句中参数值的元组。

第 25 行：调用 m_mysql 模块的 query 函数，以 SQL 和参数元组为参数，返回值为本次取到的记录数（rowcount）和二维数组的结果集。

第 26 行：定义用户信息值对象 user_vo。

第 27 行：如果取出记录不为 0，则执行第 28~31 行操作。

第 28 行：取出用户姓名。

第 29 行：取出用户邮件地址。

第 30 行：取出用户登录名。

第 31 行：取出用户密码。

第 32 行：返回用户基本信息值对象 user_vo。

至此，我们就讲解完了用户注册、登出和退出的相关功能，并且将共享密钥发送给了客户端。但是由于这里只是一个示例程序，并没有关注实现的细节，还有很多部分需要完善，例如，用户密码肯定不能直接存放在数据库中，所以数据库中的 login_pwd 应该存密码的 SHA1 摘要值，而且我们的服务请求页面只能上传一次图片等，如果提供商用级别的实现，代码量会非常大，这样会掩盖我们想要讲述的主要内容。毕竟我们在这里主要是向读者展示怎样做一个深度学习服务云平台，所以只提供了框架性的内容，具体细节读者可以自己实现。

15.4.2 客户端生成请求

完成了用户登录功能之后，用户上传完图片文件，在请求图片识别服务时，必须按照

我们的规则来生成合法的 URL，否则服务器将拒绝用户的服务请求。客户端生成请求 URL 的步骤如下。

- (1) 生成唯一标识的请求编号 req_id。
 - (2) 从 localStorage 中获取用户编号 user_id 和共享密钥 shared_key。
 - (3) 假设原始的请求为 reqUrl，则生成认证字符串为 user_id=***&req_id=*** + shared_key。
 - (4) 对认证字符串执行 SHA1 算法，生成摘要串。
 - (5) 生成新的请求 URL：reqUrl + &user_id=用户编号&req_id=请求编号&mac=摘要串。
- 下面来看看具体的代码实现，代码如下：

```
124 <script src="public/js/sha1.js"></script>
```

为了在 JavaScript 中可以对字符串计算 SHA1 摘要值，需要引入 sha1.js 库文件，在程序中调用 hex_sha1 这个函数就可以了。

在客户端，需要为每个请求添加 user_id、req_id 和 mac 参数。我们可以将这一功能包装为一个函数，放到一个公共的 JS 文件中。但是为了演示方便，这里就直接将其放在服务请求页面中。在读者的应用中，应该将这个函数放入一个全局 JS 文件中，因为几乎所有页面都会用到这个功能。

下面来看看生成合法请求的函数，代码如下：

```
369 var g_req_id = 1;
370 function generate_auth_url(reqUrl) {
371     g_req_id++;
372     var user_id = localStorage.getItem("user_id");
373     var shared_key = localStorage.getItem("shared_key");
374     var raw_str = "user_id=" + user_id + "&req_id=" + g_req_id +
375                 "&shared_key=" + shared_key
376     var mac_str = hex_sha1(raw_str)
377     return reqUrl + "&user_id=" + user_id + "&req_id=" +
378                 g_req_id + "&mac=" + mac_str;
379 }
```

第 369 行：定义全局变量 g_req_id，产生唯一的请求编号。

第 370 行：生成合法请求 URL，参数为原始的 URL。

第 371 行：将请求编号加 1。

第 372 行：从 localStorage 中取出 user_id。

第 373 行：从 localStorage 中取出 shared_key。

第 374、375 行：形成用户进行 SHA1 摘要的原始字符串。

第 376 行：对原始字符串生成 SHA1 摘要值。

第 377、378 行：返回合法的请求 URL。

第一个需要进行请求合法性认证的是图片文件上传服务，因此在调用图片上传服务时需要将原来的 URL 改为经过这个函数改造的合法 URL，需要在 chp15/public/s_mlp1.html 文件中修改如下内容：

```
157 var url = generate_auth_url('/upload/upload_mlp1?a=1'),
```

15.4.3 服务器端验证请求

服务器端收到请求后，如果不是用户注册或登录请求，则需要对请求进行认证，具体步骤如下。

- (1) 取出请求的 URL，从中找出 mac 值。
- (2) 形成认证字符串：user_id=用户编号&req_id=请求编号&shared_key=共享密钥。
- (3) 运行 SHA1 摘要算法生成摘要串。
- (4) 将生成的摘要串与 URL 中的摘要串进行比对，如果一致则说明此请求为合法请求，进行正常处理流程，如果不一致则直接拒绝该请求。

在服务器端，上传控制器收到请求后，首先进行合法性验证，如果是合法请求才允许进行文件上传操作，如果是非法请求则直接提示上传失败，代码如下：

```
13 import common.wky_auth as wky_auth

35 def upload_mlp1(self, params={}, data_file=None):
36     print('##### upload:%s' % params)
37     if not wky_auth.validate_req(params['kwargs']):
38         return {'status': 'Error'}
```

第 13 行：引入本应用通用模块 wky_auth，专门用于请求验证。

第 35 行：上传文件的消息响应函数。

第 37 行：调用 wky_auth 模块的验证函数，参数为 QueryString 中所带的参数。

第 38 行：如果请求没有通过合法性验证，则返回上传失败信息给客户端。

在客户端处理上传文件的消息响应函数中，需要识别这种情况，向用户提示文件上传失败，并直接返回，代码如下：

```
227     }).on('fileuploaddone', function (e, data) {
228         if (data.result.status != 'Ok') {
229             alert("文件上传失败！");
230             return ;
231         }
232         $("#samples_id").val(data.result.samples_id);
```

通过返回值中的 status 值是否等于“Ok”来判断文件上传是否成功，如果不成功则提示文件上传失败，并且不做任何处理直接返回。文件上传成功才显示识别图像按钮，让用户进行下一步操作。

下面来看看请求合法性验证的具体实现逻辑，在全局变量中定义服务器端与客户端的共享密钥，代码如下：

```
1 import os
2
3 dataset_dir = os.getcwd() + '/data/'
4 ann_mf_dir = os.getcwd() + '/repository/'
5 web_dir = os.getcwd()
6 upload_dir = os.getcwd() + '/upload/'
7
8 shared_key = 'wky'
```

下面来看看 wky_auth 模块，代码如下：


```

1 import hashlib
2 import app_global as ag
3
4 def validate_req(params):
5     user_id = params.get('user_id', '0')
6     req_id = params.get('req_id', '0')
7     mac = params.get('mac', 'x')
8     raw_str = 'user_id=' + user_id + '&req_id=' + req_id + \
9             '&shared_key=' + ag.shared_key
10    new_mac = hashlib.sha1(raw_str.encode('utf8')).hexdigest()
11    if new_mac == mac:
12        return True
13    else:
14        return False

```

第 1 行：引入做 SHA1 摘要所需要的库 hashlib。

第 4 行：定义请求合法性验证函数 validate_req，参数为 QueryString 中所带的参数。

第 5 行：从请求参数中解析出 user_id。

第 6 行：从请求参数中解析出唯一请求编号 req_id。

第 7 行：从请求参数中解析出客户端算出来的 SHA1 摘要。

第 8、9 行：生成用于 SHA1 算法的原始字符串。

第 10 行：重新计算 SHA1 摘要值。

第 11、12 行：如果新计算出的 SHA1 摘要值与客户传送过来的值相等，则返回请求为合法的请求。

第 13、14 行：如果新计算出的 SHA1 摘要值与客户端传送过来的值不相等，则返回请求为非法请求。

以上描述的是文件上传请求验证过程，以及图片识别请求验证，其基本流程和文件上传流程相同，在这里只列出需要改动的代码，细节内容读者可以参考文件上传部分的解释和说明。

下面来看看客户端，代码如下：

```

263 function classify_img() {
264     var ann_id = 1
265     var samples_id = $("#samples_id").val();
266     var reqUrl = "/wky/dl/mlp?cls=c_mlp&cmd=classify_img&ann_id=" + ann_id +
267     "&samples_id=" + samples_id;
268     reqUrl = generate_auth_url(reqUrl);
269     $.ajax({
270         url: reqUrl,
271         type: "GET",
272         dataType: "json",
273         success: onClassifyImgOk,
274         error: onClassifyImgError
275     });
276 }
277 function onClassifyImgOk(json) {
278     if (json.status != 'Ok') {
279         alert("图片识别失败！");
280         return ;
281     }
282     $("#classify_result").css("display", "block");
283     var result = json.result;
284     var arrs = JSON.parse(json.result);
285     var result = arrs[0];
286     $("#result").text(result);
287 }

```

第 267 行：生成合法的请求 URL。

第 277~280 行：处理服务器端返回请求不合法时的处理逻辑，这里仅显示图片识别失败，并退出程序。

读者可能有疑问，这里为什么不提示用户友好信息呢？因为非法请求通常都是恶意攻击代码，我们拒绝他们的请求，如果给的提示信息很明确，会帮助他们攻破防护措施。在这种情况下，如果想做得更好，还可以随意给他们返回一个成功的结果，这样他们就会以为请求是成功的，这就是所谓的蜜钱策略。

在多层感知器控制器中，需要进行请求合法性验证，代码如下：

```
5 import common.wky_auth as wky_auth

43 def classify_img(params):
44     kwargs = params['kwargs']
45     if not wky_auth.validate_req(kwargs):
46         return {'status': 'Error'}
47     args = params['args']
48     ann_id = kwargs['ann_id']
49     samples_id = kwargs['samples_id']
50     samples = get_samples(samples_id)
51     result = run(ann_id, samples)
52     resp = {'status': 'Ok', 'result': '%s' % result}
53     return resp
```

第 5 行：引入应用通用模块 `wky_auth`，用于请求合法性验证。

第 45、46 行：调用 `wky_auth` 模块的 `validate_req` 函数，对请求合法性进行验证。如果为非法请求则返回出错信息，只有合法请求才会调用多层感知器引擎进行图片识别工作，从而大大减少受攻击时对系统的影响。

到目前为止，我们已经完成了对请求进行合法性认证的全部过程，这样我们的深度学习云平台放到公网上后，才不会轻易地受到黑客的攻击。

15.5 异步结果处理

深度学习服务有可能算法非常复杂，或者网络比较庞大，因此在很短的时间内很可能不能产生用户想要的结果，此时我们希望云平台具有异步服务的能力，即用户先上传任务到系统的任务队列中，由后台线程逐一处理，之后再由系统通知用户来获取提交任务的结果。这样更有利于服务器进行资源分配，也有助于提高用户体验。在本节中，我们将向读者介绍这部分功能的实现原理。

15.5.1 网页异步提交

我们还以上一节中的服务请求页面为例，添加“异步请求”按钮，代码如下：

```
89 <span id="classify_btn" class="btn btn-success fileinput-button" style="
    display: none;">
90     <i class="glyphicon glyphicon-plus"></i>
91     <span>图像识别</span>
92 </span>
93 <span id="async_classify_btn" class="btn btn-success fileinput-button" s
    tyle="display: none;">
94     <i class="glyphicon glyphicon-plus"></i>
95     <span>异步识别</span>
96 </span>
```

在“图像识别”按钮下面添加“异步识别”按钮，且初始时为隐藏状态。

在页面加载时绑定单击事件响应函数，代码如下：

```
159    $('#classify_btn').bind("click", classify_img);
160    $('#async_classify_btn').bind("click", async_classify_img);
```

文件上传成功后，与“图像识别”按钮一样，显示“异步识别”按钮，代码如下：

```
241    $('#classify_btn').css("display", "block");
242    $('#classify_btn').width(100);
243    $('#async_classify_btn').css("display", "block");
244    $('#async_classify_btn').width(100);
```

当用户单击当前用户信息中的“退出”按钮时，隐藏“异步识别”按钮，代码如下：

```
378    $('#classify_btn').css("display", "none");
379    $('#async_classify_btn').css("display", "none");
```

下面来看看“异步识别”按钮单击事件响应函数，代码如下：

```
399 function async_classify_img() {
400     var ann_id = 1
401     var samples_id = $('#samples_id').val();
402     var reqUrl = "/wky/dl?cls=async_controller&cmd=async_classify_img&ann_id
    =" + ann_id + "&samples_id=" + samples_id;
403     reqUrl = generate_auth_url(reqUrl);
404     $.ajax({
405         url: reqUrl,
406         type: "GET",
407         dataType: "json",
408         success: onAsyncClassifyImgOk,
409         error: onAsyncClassifyImgError
410     });
411 }
412
413 function onAsyncClassifyImgOk(json) {
414     if (json.status != 'Ok') {
415         alert("异步识别图片失败！");
416         return ;
417     }
418     alert("异步识别图片请求已接收，请等待系统通知结果");
419 }
420 function onAsyncClassifyImgError(msg) {
421     alert("异步识别图片失败：" + JSON.stringify(msg) + "！");
422 }
```

第 399 行：定义“异步识别”按钮单击事件响应函数。

第 400 行：指定神经网络编号。

第 401 行：求出上一步操作上传图片时所对应的样本集编号。

第 402 行：生成异步识别调用 URL，重点是调用的命令是 `async_classify_img`。

第 403 行：生成系统合法的请求 URL。

第 404~410 行：发送 AJAX POST 请求。

第 413 行：异步识别成功消息响应函数。如果识别状态为失败，例如请求不合法等，则提示图片识别失败。因为此处是异步请求，因此响应状态为成功时应该提示用户以异步方式等待系统处理结果通知。

第 420~422 行：异步识别失败消息响应函数，这里只提示请求失败的原因。

15.5.2 应用队列管理模块

为了高效处理用户的请求和发送处理结果，我们将采用任务队列的方式。需要特别注意的是，这里的异步任务是基于队列的，而不是基于多线程模式的，这主要是性能方面的原因。

具体来讲，我们将在系统中建立两个队列，一个是待处理的识别任务队列 `task_q`，另一个是待发送的处理结果队列 `result_q`。系统接收到异步任务之后，将其加入识别任务队列 `task_q` 中，并立即向客户端返回请求已接收，等待处理结果的响应。

识别任务队列会对应一个任务队列处理线程，其不停地从任务队列中取出任务，并调用相应的深度学习模型，运行识别任务，产生处理结果，并将其添加到处理结果队列中。

处理结果队列同样对应一个处理结果队列管理线程，其也不停地从处理结果队列中取出处理结果，根据该结果对应的用户，向用户指定的联系方式发送通知信息。

具体的系统架构如图 15.8 所示。

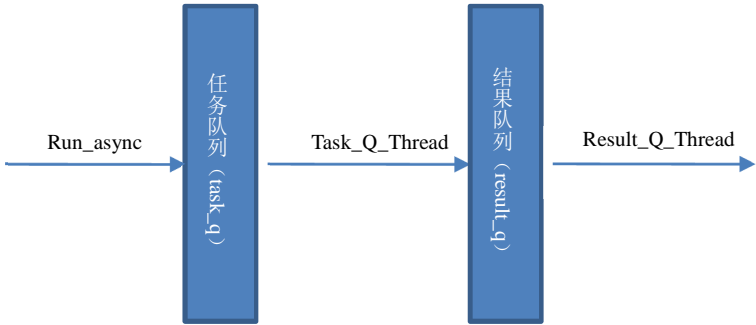


图 15.8 任务队列管理系统架构图

下面来看看具体的实现细节，首先在 `app_global` 中定义队列和队列线程的全局变量，代码如下：

```
1 import os
2
3 dataset_dir = os.getcwd() + '/data/'
4 ann_mf_dir = os.getcwd() + '/repository/'
5 web_dir = os.getcwd()
6 upload_dir = os.getcwd() + '/upload/'
7
8 shared_key = 'wky'
9
10 global task_q
11 global result_q
12 global task_q_thread
13 global result_q_thread
```

- 第 10 行：定义全局变量表示识别任务队列。
 - 第 11 行：定义全局变量表示处理结果队列。
 - 第 12 行：定义全局变量表示识别任务队列所对应的任务队列处理线程。
 - 第 13 行：定义全局变量表示处理结果队列所对应的处理结果处理线程。
- 接下来看看线程管理类，代码如下：

```

1 import threading
2 import queue
3 import app_global as ag
4 import common.task_queue as tq
5 import common.result_queue as rq
6
7 def init_wky_queues():
8     print('初始化任务队列、结果队列，启动任务队列处理线程和结果队列处理线程')
9     ag.task_q = queue.Queue(maxsize=10)
10    ag.result_q = queue.Queue(maxsize=10)
11    ag.task_q_thread = tq.Task_Q_Thread()
12    ag.task_q_thread.start()
13    result_q_thread = rq.Result_Q_Thread()
14    result_q_thread.start()

```

第 7 行：定义队列系统初始化函数。

第 9 行：初始化任务队列为最大值是 10 的队列对象。

第 10 行：初始化处理结果队列为最大值是 10 的队列对象。

第 11 行：生成任务队列对应的任务队列处理线程对象。

第 12 行：启动任务队列处理线程。

第 13 行：生成处理结果队列对应的处理结果队列处理线程对象。

第 14 行：启动处理结果队列处理线程。

15.5.3 任务队列

系统接收到异步请求之后，不是立即处理，而是先将其加入任务队列中，由任务队列对应的任务队列处理线程从任务队列中取出任务，再执行该任务，将结果添加到处理结果队列中，然后从任务队列中取下一个任务，如此循环往复执行。

下面来看看任务队列的具体实现，代码如下：

```

1 import threading
2 import queue
3 import app_global as ag
4 import controller.c_mlp as c_mlp
5
6 class Task_Q_Thread(threading.Thread):
7     def __init__(self):
8         threading.Thread.__init__(self)
9         print('初始化任务队列处理线程')
10
11    def run(self):
12        print('启动任务队列处理线程')
13        params = ag.task_q.get(block=True)
14        while params:
15            print('获取任务队列中的任务:%s' % params)
16            result = c_mlp.classify_img(params)
17            result['user_id'] = params['kwargs']['user_id']
18            result['req_id'] = params['kwargs']['req_id']
19            result['order_id'] = result['user_id'] + '_' + result['req_id']
20            print('加入到结果队列:%s' % result)
21            ag.result_q.put(result)
22            params = ag.task_q.get(block=True)

```

第 1 行：引入线程相关库。

第 2 行：引入队列库。

第 6 行：定义任务队列对应的任务队列处理线程类。

第 7~9 行：定义构造函数。

第 11 行：定义线程启动方法。

第 13 行：以阻塞方式从任务队列中读出最先进入队列的任务，如果任务队列为空，则等到任务队列有任务时才返回。

第 14 行：无限循环地从任务队列中取出需要执行的任务。

第 16 行：根据取出的任务，选取适当的深度学习模型控制器类，调用其识别方法，并记录处理结果。在本例中，直接使用多层感知器模型。在实际项目中，我们可以根据请求的内容，选择合适的深度学习算法来处理。

第 17 行：在处理结果中添加用户编号信息。

第 18 行：在处理结果中添加请求编号信息。

第 19 行：在处理结果中添加订单号信息，供用户跟踪请求的处理进度。

第 21 行：将处理结果添加到处理结果队列中。至此一个异步请求的处理过程就结束了。

第 22 行：重新从任务队列中读取下一个任务，重复第 15~21 行的操作，如果任务队列为空，则阻塞当前线程，直到任务队列中有任务为止。

需要注意的是，我们在这里只是向读者演示异步任务的处理流程，因此深度学习模型控制器是固定的多层感知器模型，在读者的应用系统中可以根据神经网络编号 `ann_id` 动态选择深度学习模型的控制器，或者采用设计模式中的工厂模式生成所需的深度学习模型控制器类对象。

15.5.4 结果队列

处理结果队列对应的处理结果线程，从处理结果队列中取出处理结果，根据处理结果的用户信息向用户指定的联系方式发送通知信息，通知用户查收任务的处理结果。具体实现代码如下：

```
1 import threading
2 import queue
3 import app_global as ag
4
5 class Result_Q_Thread(threading.Thread):
6     def __init__(self):
7         threading.Thread.__init__(self)
8         print('初始化结果队列线程')
9
10    def run(self):
11        print('启动结果队列处理线程')
12        result = ag.result_q.get(block=True)
13        while result:
14            print('发送结果: %s' % result)
15            result = ag.result_q.get(block=True)
```

第 1 行：引入线程相关类。

第 2 行：引入队列相关类。

第 5 行：定义处理结果队列处理线程类，其为 `threading` 子类。

第 6~8 行：定义构造函数。

第 10 行：定义线程启动函数。

第 12 行：以阻塞方式从结果队列中取出处理结果，如果没有则阻塞当前线程执行。

第 13 行：发送处理结果的无限循环。

第 14 行：先根据用户信息查找用户的联系方式，再将处理结果发送给用户，并且通知用户“处理结果已经发送，请及时接收”。

第 15 行：从处理结果队列中取下一条处理结果，继续执行这一循环发送过程。如果处理结果队列为空，则阻塞线程执行，直到有处理结果进入队列为止。

15.5.5 异步请求处理流程

客户端的异步请求任务会发送到异步任务控制器类，将请求直接添加到任务队列中，并立即向客户端返回“请求已经正确接收，请等待处理结果”的响应。

异步请求控制器类代码如下：

```
1 import common.wky_auth as wky_auth
2 import app_global as ag
3
4 def async_classify_img(params):
5     kwargs = params['kwargs']
6     if not wky_auth.validate_req(kwargs):
7         return {'status': 'Error'}
8     print('异步任务控制器接收到新的异步任务')
9     ag.task_q.put(params)
10    resp = {'status': 'Ok'}
11    return resp
```

第 1 行：引入请求认证模块，因为异步请求也需要对请求进行认证。

第 4 行：定义图片识别异步任务接口。

第 5 行：取出请求 QueryString 中的参数 kwargs。

第 6、7 行：对请求进行验证，如果是非法请求，直接返回出错信息。

第 9 行：将请求加入任务队列中。

第 10、11 行：返回成功接收到请求的响应。

我们需要在程序启动时就启动异步队列系统，代码如下：

```
1 import sys
2 sys.path.append('./lib/Theano')
3 sys.path.append('./lib/cherrypy')
4 import os
5 import theano
6 import cherrypy
7 import app_global as ag
8 import model.m_mysql as db
9 import controller.c_mlp as mlp
10 import app_web as app_web
11 import common.wky_queues as wqs
12
13 if __name__ == '__main__':
14     print('starting up...')
15     db.init_db_pool()
16     wqs.init_wky_queues()
17     app_web.startup()
18     db.rdb_pool_cleaner.join()
19     db.wdb_pool_cleaner.join()
20     ag.task_q_thread.join()
21     ag.result_q_thread.join()
```

第 11 行：引入队列管理模块。

第 16 行：初始化队列管理系统。

程序的运行结果如下：

```

1 starting up...
2 初始化任务队列、结果队列，启动任务队列处理线程和结果队列处理线程
3 初始化任务队列处理线程
4 启动任务队列处理线程
5 初始化结果队列处理线程
6 启动结果队列处理线程
7 192.168.1.7 - - [13/Jan/2017:09:12:54] "GET /web/pages?cmd=show_s_mlp_1 HTTP
  /1.1" 200 16461 "" "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537
  .36 (KHTML, like Gecko) Chrome/54.0.2840.87 Safari/537.36"
8 .....
9 ##### upload: {'kwargs': {'user_id': '21', 'mac': 'b4e825a5dbd67557b7ee
  0d11a2e948ce6ed3c531', 'req_id': '2', 'a': '1'}, 'args': {}}
10 192.168.1.7 - - [13/Jan/2017:09:13:01] "POST /upload/upload_mlp1?a=1&user_id
  =21&req_id=2&mac=b4e825a5dbd67557b7ee0d11a2e948ce6ed3c531 HTTP/1.1" 200 48 "
  http://192.168.1.16:8090/web/pages?cmd=show_s_mlp_1" "Mozilla/5.0 (Windows N
  T 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.8
  7 Safari/537.36"
11 <module 'controller' (namespace)>
12 .....
13 异步任务控制器接收到新的异步任务
14 192.168.1.7 - - [13/Jan/2017:09:13:03] "GET /wky/dl?cls=async_controller&cmd
  =async_classify_img&ann_id=1&samples_id=31&user_id=21&req_id=3&mac=b23497fa1
  6a953eeb9aaa3bb8e7ade1aa40f4191 HTTP/1.1" 200 16 "http://192.168.1.16:8090/w
  eb/pages?cmd=show_s_mlp_1" "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWe
  bKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.87 Safari/537.36"
15 获取任务队列中任务: {'kwargs': {'user_id': '21', 'mac': 'b23497fa16a953eeb9aa
  a3bb8e7ade1aa40f4191', 'req_id': '3', 'cls': 'async_controller', 'cmd': 'asy
  nc_classify_img', 'samples_id': '31', 'ann_id': '1'}, 'args': {}}
16 create MlpMnistEngine
17 加入结果队列: {'user_id': '21', 'req_id': '3', 'status': 'Ok', 'result': '
  [2]', 'order_id': '21_3'}
18 发送结果: {'user_id': '21', 'req_id': '3', 'status': 'Ok', 'result': '[2]',
  'order_id': '21_3'}

```

15.6 神经网络持续改进

深度学习网络相当于一个黑盒子，我们很难对一个深度学习网络进行理解和调优，对一个已经投入生产的深度学习网络，怎样改进其性能是一个难题。在这一节中，我们将向读者介绍三种相对有效的方法，希望能对读者有所启发。

15.6.1 应用遗传算法

遗传算法（Genetic Algorithm, GA）是与神经网络齐名的算法，也是解决优化问题的有效方法。遗传算法是模拟生物进化过程的自然选择机制，采用现代遗传学机理知识，对生物进化过程建立计算机模型，通过模拟自然进化过程，搜索最优解的方法。

遗传算法从代表问题可能解的一个种群开始，种群中的个体具有特定的基因编码。个体的遗传信息用染色体表示，染色体上有多个基因，每个基因的表现形式被称为基因型，其会对应一到多个表现型，即模型的外在性能。在具体实现中，基因和基因型通常用二进

制编码的字符串来表示。种群个体间会进行繁殖，在繁殖过程中，基因会在遗传算子作用下进行组合交叉和变异，从而产生新一代种群和个体。我们针对个体有一个适应度函数，可以计算每个个体的适应度。在每次繁殖结束后，我们都会根据个体适应度，选择适应度好的个体，淘汰适应度差的个体。经过很多代之后，种群中的个体将都是适应度好的个体。而我们所选择的适应度，就是希望达到的目标，通过这一过程就可以找到问题的最优解。

要改进神经网络性能，我们自然会想到和神经网络齐名的遗传算法，是否可以将两者结合起来呢？这是一个比较有争议的观点。很多研究人员，包括深度学习之父 Hinton 等人，都认为遗传算法是不能应用于神经网络训练学习过程的。

遗传算法的优点在于全局搜索的能力很强，但是局部搜索效率低；而神经网络正好与之相反，它的优势在于局部搜索能力较强，但是全局搜索能力较差，二者似乎有可以结合起来的潜力。

基于上述这两种算法的特点，我们提出了遗传算法与神经网络相结合的模式。仍以多层感知器为例，我们以所有连接权值的二进制表示作为基因表示方式，设定初始种群数量为 g ，利用随机数初始化这 g 个神经网络。我们先用传统的多层感知器算法对这些网络进行训练，可以设定最大迭代次数为终止条件，再用最终网络在验证样本集或测试样本集上的误差作为适应度函数，淘汰其中 r 个个体。将剩下的个体按照标准遗传算子进行交叉和变异操作，重新生成新的连接权值，产生新一代 g 个个体。我们再对这些网络利用多层感知器算法进行训练，直到达到终止条件。这时同样将这些网络在验证样本集或测试样本集上的误差作为适应度函数，淘汰 r 个个体，再进行下一代繁殖。以此类推，直到达到最终的理想效果为止。

根据我们的实际应用经验，将遗传算法和神经网络相结合时，神经网络的学习率应该适当减小，因为我们需要的是神经网络的局部搜索能力，同时动量项和权值调整项也需要适当取更小的值。而对于遗传算法，我们可以将变异率取得适当大一些，这样更能保证全局搜索的可行性。

15.6.2 重新训练

对于深度学习网络而言，最重要的性能就是其泛化能力，即遇到新样本时可以得出正确结论的能力。但是由于欠拟合和过拟合现象的存在，如果我们的模型比实际问题要简单，那么训练样本集上将会有较大的误差。相反，如果我们的模型比实际问题复杂很多，就又会出现在训练样本集上的误差很小，而验证样本集和测试样本集上的误差却非常大的现象。这就是通常所说的机器学习中的 **Bias/Variance** 困境，优化其中之一将使另外一个恶化，我们必须在二者间取得平衡。

目前有很多技术来帮助我们达到这一目的，例如我们在前面各章的示例中广泛采用的早期停止技术，就是试图在验证样本集误差出现恶化前停止训练过程，从而达到最大化现有网络泛化能力的目的。另外，比较有效的方式是神经元的 **Dropout**，通过这种技术有效减

少了需要学习的参数和解的搜索空间，也有助于提高模型的泛化能力。这一领域是目前深度学习的热门研究领域，技术发展非常快，读者可以重点关注一下这个领域。

对于已经投入生产环境的深度学习网络而言，还有一个非常重要的有利条件，我们可以用其来提高深度学习网络的泛化能力。深度学习网络在实际运行过程中，会接收到非常多的新样本数据，而深度学习网络的泛化能力不高，一个重要原因就是训练样本集不够大，或者训练样本集代表性不足，而在实际运行过程中，收集到的大量样本无疑是一笔宝贵的财富，我们必须加以利用。

在本章中，我们对用户请求的数据都进行了记录，我们可以通过这种方式收集足够多的样本数据。当我们收集到足够多的样本数据后，就可以通过批量方式对这些数据进行标注，并将标注后的数据加入训练样本集中。隔一段时间后，对现有网络在新的训练样本集上进行训练，通过在同样的验证样本集和测试样本集上进行验证，比较训练前后在验证样本集和测试样本集上的误差，如果有改进则将新的深度学习网络上线，替代原有网络。

15.6.3 生成式对抗网络

生成式对抗网络是深度学习领域的一个重要趋势，这种网络由两部分组成，生成式网络部分通过采样产生逼真的伪造样本，并将这些伪造样本与真实样本都输入到模式识别网络，模式识别网络的任务就是识别出哪些是真实样本，哪些是伪造样本。生成式网络部分和模式识别网络部分并行学习，这样生成式网络产生的伪造样本越来越真实，而模式识别网络的识别能力也越来越强。这有点像自然界中的捕食者与被捕食者，在互相的生存竞争中练就了非凡的本领。

受到这种理念的启发，如果我们没有条件获得大量样本，也可以利用生成式网络来产生样本。我们在 10.2 节讨论受限玻尔兹曼机时，将其用于 MNIST 手写数字识别任务，最终训练完成后，我们可以运行采样程序生成仿真的样本，如图 15.9 所示。



图 15.9 利用受限玻尔兹曼机生成仿真样本

MNIST 手写数字识别数据集上的原始图片如图 15.10 所示。

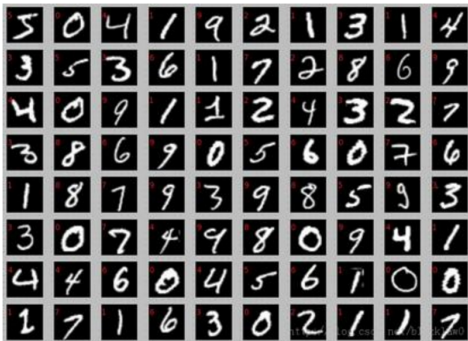


图 15.10 MNIST 手写数字识别真实样本

可以看到，通过受限玻尔兹曼机生成的仿真样本，与 MNIST 手写数字识别的真实样本区别很小，基本可以作为训练样本使用。

如果将受限玻尔兹曼机生成的仿真样本，先通过人工来选择合适的样本，并进行人工标注，将其合并到训练样本集，然后重新训练我们的深度学习网络。训练过程结束后，我们与之前的深度学习网络进行比较，如果验证样本集和测试样本集上的误差有改进，则可以考虑在实际中采用新的网络参数。

后 记

本书成书的初衷有三点：首先，系统讲述深度学习的中文图书需求非常大；其次，在深度学习技术大热的当下，我们需要的是一本能够帮助广大工程技术人员，尤其是程序开发人员快速进入深度学习技术领域的书；最后，也是最重要的一点，在当前关于深度学习的书籍中，一类是以数学原理为主、全书充满数学公式的数学书，一类是深度学习开源框架的使用教程，只讲述对 TensorFlow、Caffe、Torch 或 Theano 等的使用，对原理部分涉及很少，这两种书籍实际上都有不足。对于第一类书籍，我们可能很难看懂其中的内容，而且即使看懂了其中的内容，也会苦于不知道如何用自己熟悉的编程语言来实现；对于第二类书籍，我们可以很容易运行 Hello World 级程序，但是然后要怎么做，怎样把这些技术框架应用到实际项目中，并且持续进行优化，就无从下手了。本书就是想填补深度学习数学知识和深度学习网络实现之间的鸿沟，使读者不仅了解深度学习背后的数学知识，同时掌握基于开源框架的实现技术，希望读者不仅能够迅速地将所学到的深度学习技能应用到自己的实际项目中去，而且具有跟踪顶级期刊最新文章的能力，可以持续更新自己的深度学习知识和技能体系。

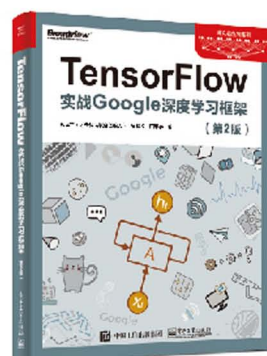
当然，深度学习知识体系非常庞大，而且还处于快速发展之中，所以虽然我竭尽全力使本书能够作为深度学习从入门到精通的指导书，但是还是存在很多不足之处，恳请大家多提宝贵意见，以利于改进和提高，共同进步。

参考文献

- [1] Andren Ng.(2003) CS229 Machine Learning Class. Stanford University.
- [2] Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I. J., Bergeron, A., Bouchard, N., and Bengio, Y.(2012). Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop.
- [3] Bengio, Y.(2009). Learning deep architectures for AI. Now Publishers.
- [4] Bengio, Y. and Delalleau, O.(2009). Justifying and generalizing contrastive divergence. Neural Computation, 21(6), 1601-1621.
- [5] Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H.(2007). Greedy layer-wise training of deep networks. In NIPS'2006.
- [6] Bengio, Y., Lamblin, P., Popovici, D., and Larochelle, H. (2007a). Greedy layer-wise training of deep networks. In NIPS'2006 .
- [7] Bengio, Y., Yao, L., Alain, G., and Vincent, P. (2013b). Generalized denoising autoencoders as generative models. In NIPS'2013 .
- [8] Carreira-Perpinan, M. A. and Hinton, G. E. (2005). On contrastive divergence learning.
- [9] In R. G. Cowell and Z. Ghahramani, editors, AISTATS'2005 , pages 33-40. Society for Artificial Intelligence and Statistics.
- [10] Courville, A., Desjardins, G., Bergstra, J., and Bengio, Y. (2014). The spike-and-slab RBM and extensions to discrete and sparse data distributions. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 36(9), 1874-1887.
- [11] Deeplearning.net. [http://http://deeplearning.net/tutorial/contents.html](http://deeplearning.net/tutorial/contents.html).
- [12] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. In CVPR09 .
- [13] Erhan, D., Bengio, Y., Courville, A., Manzagol, P., Vincent, P., and Bengio, S. (2010). Why does unsupervised pre-training help deep learning? J. Machine Learning Res.
- [14] Goodfellow, I. J., Courville, A., and Bengio, Y. (2011). Spike-and-slab sparse coding for unsupervised feature discovery. In NIPS Workshop on Challenges in Learning Hierarchical Models.
- [15] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S.,

- Courville, A., and Bengio, Y. (2014c). Generative adversarial networks. In NIPS'2014 .
- [16] Graves, A. (2012). Supervised Sequence Labelling with Recurrent Neural Networks. Studies in Computational Intelligence. Springer.
- [17] Hinton, G. E., Osindero, S., and Teh, Y. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, 18, 1527–1554.
- [18] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735–1780.
- [19] Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks* , 2, 359–366.
- [20] Hubel, D. H. and Wiesel, T. N. (1959). Receptive fields of single neurons in the cat's striate cortex. *Journal of Physiology* , 148, 574–591.
- [21] Hubel, D. H. and Wiesel, T. N. (1962). Receptive fields, binocular interaction, and functional architecture in the cat's visual cortex. *Journal of Physiology (London)*, 160, 106–154.
- [22] Karpathy, A. and Li, F.-F. (2015). Deep visual-semantic alignments for generating image descriptions. In CVPR'2015 . arXiv:1412.2306.
- [23] Kavukcuoglu, K., Sermanet, P., Boureau, Y.-L., Gregor, K., Mathieu, M., and Le-Cun, Y. (2010b). Learning convolutional feature hierarchies for visual recognition. In NIPS'2010 .
- [24] Krizhevsky, A., Sutskever, I., and Hinton, G. (2012a). ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25 (NIPS'2012)*.
- [25] Krizhevsky, A., Sutskever, I., and Hinton, G. (2012b). ImageNet classification with deep convolutional neural networks. In NIPS'2012 .
- [26] Mao, J., Xu, W., Yang, Y., Wang, J., Huang, Z., and Yuille, A. L. (2015). Deep captioning with multimodal recurrent neural networks. In ICLR'2015 . arXiv:1410.1090.
- [27] Poole, B., Sohl-Dickstein, J., and Ganguli, S. (2014). Analyzing noise in autoencoders and deep networks. CoRR, abs/1406.1831.
- [28] Sainath, T., rahman Mohamed, A., Kingsbury, B., and Ramabhadran, B. (2013). Deep convolutional neural networks for LVCSR. In ICASSP 2013 .
- [29] Schuster, M. and Paliwal, K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11), 2673–2681.
- [30] Sermanet, P., Chintala, S., and LeCun, Y. (2012). Convolutional neural networks applied to house numbers digit classification. CoRR, abs/1204.3968.
- [31] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15, 1929–1958.
- [32] Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., and Manzagol, P.-A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local

- denoising criterion. *J. Machine Learning Res.*, 11.
- [33] Wager, S., Wang, S., and Liang, P. (2013). Dropout training as adaptive regularization. In *Advances in Neural Information Processing Systems 26*, pages 351–359.
- [34] Wan, L., Zeiler, M., Zhang, S., LeCun, Y., and Fergus, R. (2013). Regularization of neural networks using dropconnect. In *ICML'2013*.
- [35] Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. (2008). Extracting and comz.
- [36] 赵永科. 深度学习：21 天实战 Caffe[M]. 北京电子工业出版社，2016.



深度学习算法实践 (基于Theano和TensorFlow)

当前关于深度学习的书籍有很多。一方面,有一部分书籍介绍流行算法和网络架构的实现,基于开源深度学习框架(如TensorFlow),结合常用数据集(如MNIST、ImageNet),实现图像识别等典型应用,读者可以很容易地做出一个演示系统,但是对于算法实现原理和调参思路却很难有深刻的理解,很难将书中所说应用到实践中去。另一方面,有一部分书籍侧重于算法原理的讲解,对数学公式推导和定理证明非常重视,但是对具体算法实现介绍得很少,使读者对理论似懂非懂,对算法的具体实现感到无从下手。本书以普通本科生都能看懂的数学原理,深入浅出地讲解了常用算法,同时基于常用的深度学习框架Theano和TensorFlow对这些算法的实现方法进行了详细的讲解,使读者不仅可以掌握算法的数学原理,还可以掌握基于Theano及TensorFlow的实现技术,从数学原理上掌握调参方法的思路和原理,为读者将书中所学知识应用到自己的项目实践打下了坚实的基础,是一本不可多得的深度学习自学参考书。

——黄文涛

中国电子科技集团创新院人工智能研究室首席科学家
西安电子科技大学人工智能学院客座教授
约翰霍普金斯大学医学院研究员、博士后

当前深度学习在各行各业的应用越来越广,因此越来越多的岗位要求深度学习方面的应用知识。深度学习在医学研究领域同样得到深入的应用,中科院苏州生物医学工程技术研究所迫切需要精通深度学习理论及应用的专业人士加入我们的团队,但是在实践过程中,我们发现很多人在如何应用机器学习解决实际问题方面,存在要么对算法数学原理理解不深入,要么对程序实现技术缺乏经验的现象。这本书从基本数学原理出发,详细讲述了当前流行算法多层感知器(MLP)、卷积神经网络(CNN)、递归神经网络(RNN)、长短时记忆网络(LSTM)、受限玻尔兹曼机(RBM)、深度信念网络(DBN)背后的数学原理,同时详细讲解了基于Theano和TensorFlow的实现技术。闫涛不仅对深度学习算法实现技术进行了详细描述,为了能够组成一个基于深度学习算法的完整应用系统,在书中最后一部分,应用服务器开发技术,以MNIST手写数字识别为例,向读者演示了一个完整的深度学习应用系统的实现技术,这一点在同类书籍中还是非常少见的。本书对于初学者来说是一本很好的入门读物,对于广大的机器学习从业者来说也是一本很好的参考书。对于想要将深度学习算法应用到自己实际项目中的读者,这本书更是提供了一套完整的实现技术,读者可以基于书中的内容,在短时间内搭建起属于自己的深度学习应用系统。

——陈晓禾

中科院苏州生物医学工程技术研究所百人计划专家
中科院苏州生物医学工程技术研究所电子室主任



博文视点Broadview



@博文视点Broadview



策划编辑:付睿
责任编辑:牛勇
封面设计:吴海燕

上架建议:人工智能/深度学习

ISBN 978-7-121-33793-2



定价:109.00元